Inference of Upper-Bounds on the Expected Cost of Probabilistic Programs

Inferencia de cotas superiores sobre el coste esperado de programas probabilistas

Alicia Merayo Corcoba

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Trabajo de Fin de Máster

Curso 2018/2019

Convocatoria: **Junio de 2019**
Calificación: **Matrícula de honor (10)**

Director:

Samir Genaim

Acknowledgements

There are too many people to say thanks so let us summarize a bit to avoid you get bored.

Thanks to my family, I love you, you know it. Martín, maybe you are too small to know it now, but your auntie loves you. I promise we will go to the zoo.

Thanks to COSTA group, especially Elvira and Samir, thanks for these past years and the ones that are coming. Pablo, the bibliography of this work and I want to say thanks to you. Jesús, it has been useful to see that I was not the only one desperate about Farkas lemma. Miguel, please come back from China.

People in my life that have not felt that they have been thanked in the previous paragraphs. If you are a person I should say thanks to, thank you.

Jo Brand said once "Anything is good if it is made of chocolate" and although this work is not a chocolate cake, it has been made eating chocolate and it must be a valuable asset. So, finally, let us thank chocolate, nothing would have been the same without it.

Inference of Upper-Bounds on the Expected Cost of Probabilistic Programs - Abstract

Resource usage analysis (a.k.a. cost analysis) aims at *statically* determining the number of resources required to safely execute a given program. A *resource* can be any quantitative aspect of the program or its environment, such as memory consumption, execution steps, energy, etc. Over the past decade several cost analysis frameworks, for different programming languages, have been developed and they can infer precise closed-form upper-bound (resp. lower-bound) functions on the *worst-case* (resp. *best-case*) cost. However, some algorithms and problems involve probabilistic choices for which these notions of cost are not adequate. A well-known example is that of *randomized algorithms*, where randomized decisions are used to make the solving of computationally hard problems efficient. Besides, some problems can be precisely described only using probabilities. For example, estimating the number of attempts needed to transmit n packets of data over the network, knowing that transmission might fail with probability p. The adequate notion of cost for such scenarios is the *expected cost* which is, roughly, the sum of multiplying the cost of each possible trace by the probability that such a trace is produced.

The goal of this work is to develop techniques for automatically inferring the expected cost of imperative probabilistic programs, using cost relations which are a formalism that generalizes classical recurrence equations and allows modeling the cost of complex programs. For this, we extend the definition of cost relations to model the expected cost, describe how to translate probabilistic programs into cost relations; and develop techniques for solving these cost relations into closed-form bounds on the expected cost. We also report on a corresponding implementation.

Keywords

Probabilistic programs, expected cost, cost relations, cost analysis.

Inferencia de cotas superiores sobre el coste esperado de programas probabilistas - Resumen

El análisis de consumo de recursos (en adelante, análisis de coste) busca determinar *estáticamente* los recursos necesarios para ejecutar un programa de forma segura. Un *recurso* es cualquier aspecto cuantitativo del programa, como el consumo de memoria o el número de pasos en una ejecución. Durante la última década se han desarrollado análisis de coste en diversos lenguajes de programación capaces de inferir funciones precisas que acotan superiormente (resp. inferiormente) el consumo en el caso peor (resp. mejor). Sin embargo, hay veces en que tenemos elecciones probabilistas que invalidan el uso de estos conceptos. Por ejemplo, en los *algoritmos probabilistas* las elecciones aleatorias permiten resolver más eficientemente problemas computacionales complejos. Además, hay problemas que sólo se pueden describir de manera precisa mediante probabilidades. Por ejemplo, la estimación del número de intentos necesarios para transmitir n paquetes de datos en una red con probabilidad de fallo p. La noción adecuada de coste para estos casos es el *coste esperado*, que se define como la suma ponderada del coste de cada posible traza con respecto a su probabilidad.

El objetivo de este trabajo es desarrollar técnicas de inferencia automática del coste esperado de programas probabilistas mediante el uso de relaciones de coste, una generalización de las relaciones de recurrencia clásicas, para modelar los programas. En particular, extenderemos las relaciones de coste para modelar el coste esperado, describiremos cómo traducir los problemas probabilistas a estas relaciones de coste y describiremos cómo resolver estas relaciones de coste para obtener funciones que acoten superiormente el coste esperado. Por último, remarcaremos la correspondiente implementación del trabajo desarrollado.

Palabras clave

Programas probabilistas, coste esperado, relaciones de coste, análisis de coste.

Table of Contents

CHAPTER 1

Introduction

Every program has functionality that it is built to perform, and an environment with which it interacts. A *resource* in this context can be any quantitative aspect of the program or the environment, such as runtime, memory allocated, energy consumed, execution steps performed, etc. Estimating the number of resources that a program might consume is important to, among other things, guarantee that it will execute without running out of resources, which might have drastic consequences depending on the environment in which it executes. This is exactly the kind of problem that the field of *resource usage analysis* (a.k.a. *cost analysis*) deals with.

Cost analysis techniques can be classified into two main approaches: *dynamic* and *static*. The dynamic approach is typically based on simulations, where the program under consideration is executed on some input data, not in its natural environment but rather in a simulated one, and the amount of resources consumed is measured by different means. This approach is typically easy to implement and is precise for the particular input (or set of inputs) on which the program is simulated, however, it does not provide any guarantee on the resource consumption when the program is executed on a different input. Unlike the dynamic approach, the static approach estimates the

resource consumption without executing the program, i.e., just by analyzing its source code, and it typically guarantees that these estimations are valid for *all* possible executions. Static approaches are more elaborated than dynamic ones, as they build on some mathematical foundations that guarantee soundness.

The outcome of a static cost analyzer is typically a function that maps input values to the cost of corresponding executions. Since the problem is clearly undecidable, these functions do not describe the exact cost but rather an upper-bound on the *worst-case* cost or a lower-bound on the *best-case* cost. They might be given in an asymptotic form as well using notations such as Big O and Big Omega. In addition, when the input data is not numerical, e.g., a data structure or an array, the corresponding data is typically abstracted to some numerical measure that is called its size, e.g., the size of an array, the length of a list, the depth of a tree, etc. For example, consider the following Java program that implements the Insertion Sort algorithm:

```
void sort(int arr[]) {
    for (int i = 1; i < arr.length; ++i) {
        int j = i - 1;
        while (j >= 0 && arr[j] > arr[i]) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = arr[i];
    }
}
```

When counting the number of times the condition of the inner loop is executed, a cost analyzer would return the upper-bound function $sort(n) = \frac{n \cdot (n-1)}{2}$ and the lower-bound function $sort(n) = n$, where n refers to the length of the input array. It might also give them an asymptotic form such as $O(n^2)$ in the worst-case and $\Omega(n)$ in the best-case.

Over the past decade several cost analysis frameworks, for different programming languages, have been developed. They can infer precise upper-bounds on the *worst-*

case cost and lower-bounds on the *best-case* cost. Early work on cost analysis [43, 25, 41, 34, 17, 23, 23, 9, 40] started by automating the classical technique used in (manual) complexity analysis, which is based on modeling the cost of a program using *recurrence equations* and then solving them into closed-form functions using off-the-shelf computer algebra systems. The applicability of these techniques is limited, since modeling the exact cost using recurrence equations is not always possible, in particular for programs that include some kind of nondeterminism that is either explicit in the programming language or comes from abstractions (e.g., abstracting data structures to their sizes, abstracting nonlinear arithmetic to linear arithmetic, etc). To overcome these limitations, the notion of *cost relations* was introduced in [3], which generalizes recurrence equations to allow for nondeterminism. Several practical techniques for solving cost relations into closed-forms upper-bound and lower-bound functions have been developed [3, 5, 6, 21, 20]. Cost relations are the bases of cost analyzers such as `COSTA` [4] and `SACO` [2]. There are other cost analysis techniques that are not based on the use of recurrence equations or cost relations (see Section 5.1). In this work, we are interested in those based on cost relations.

1.1 Expected cost of probabilistic programs

Worst-case and *best-case* cost analyses have many practical and important applications, however, there are scenarios (algorithms, problems, etc.) that involve probabilistic choices and for which these notions are not adequate.

A well-known example is that of the important field of *randomized algorithms*. These are algorithms that use randomized decisions (involving probabilities) to make the solving of computationally hard problems efficient. Due to the use of randomized decisions, these algorithms might produce wrong answers, however, they provide guarantees that wrong answers can be produced with a very low probability. The runtime complexity of these algorithms typically depends on the probabilities induced by the

randomized decisions. A famous example of such an algorithm is the *Miller-Rabin Randomized Primality Test*, which is a randomized algorithm that checks if a number is prime in polynomial-time.

Apart from randomized algorithms, some problems can be precisely modeled only using probabilities. For example, consider a program that transmits packets of data over the network and that, due to network failures, the transmission of a packet might fail and it has to be transmitted again. Taking the environment into account, i.e., the physical network, such behavior can be precisely quantified using probabilities, for example, the probability of a successful transmission is $\frac{3}{4}$ and that of a failed transmission is $\frac{1}{4}$. The goal is to estimate the number of attempts required to successfully transmit n packets. This problem can be modeled precisely using the following probabilistic program:

```
while ( n > 0 ) {
  tick(1);
  n = n−1; ⊕3/4 skip;
}
```

where $\oplus$ is a probabilistic choice operator and **tick**(1) is an instruction that represents resource consumption (a transmission attempt). Note that the left-hand (resp. right-hand) side of the operator $\oplus$ corresponds to a successful (resp. failed) transmission.

The resource consumption of this program represents the number of attempts required to transmit n packets. An easy and sound way to estimate this number is to consider the probabilistic choice as a nondeterministic choice, and then analyze the program using a *worst-case* cost analyzer. This, however, would not even obtain an upper-bound since the loop above is then considered nonterminating because the branch of **skip** can be taken continuously, even if in practice the probability of such execution is 0. The adequate notion of cost for this setting, i.e., in the presence of such probabilistic operations, is the *expected cost*, which considers the cost of all traces but taking into account the probability of each one as well. Roughly, the expected cost

is the sum of multiplying the cost of each possible trace by the probability that such a trace is produced. For the program above, the expected cost is $\frac{4}{3} \cdot n$.

Automatic *expected cost* analysis for probabilistic programs, mainly its practical side, is relatively a new research field and has been recently considered in several works [28, 33, 7, 13, 42]. A breakthrough that triggered practical research in this field is part of [28], where the expected cost was formalized using a weakest precondition calculus that can handle nondeterministic programs as well, which was a major difficulty until then.

1.2 Objectives and contributions

The goal of this work is to explore the use of cost relations, that we mentioned above, in the context of expected cost analysis. This includes extending the definition of cost relations to allow modeling the expected cost of nondeterministic probabilistic programs, and to develop corresponding techniques to solve them into closed-form upper-bound functions. In particular, we would like to follow a similar methodology to that used for worst-case cost analysis, i.e., start from recurrence equations and generalize them to handle complex probabilistic programs.

The contributions that we make in this work in order to achieve this goal are the following:

- We extend the definition of cost relations of [3] to model probabilistic branching.
- We develop a transformation that translates a given imperative probabilistic program into a cost relation that captures its expected cost.
- We suggest a technique for solving these cost relations into *linear* closed-form upper-bound functions.
- We report on a preliminary implementation of an expected cost analyzer that

is written in Python.

A preliminary version of this work [31] has been presented at the *International Workshop on Termination, WST 2018*.

We note that the above goal is the short term goal that we deal with in this work, however, our long term goal is to use this experience in order to add support for the inference of expected cost in our cost analyzer `SACO` [2], which currently infers upper-bounds on the worst-case cost of `ABS` programs [27] — an abstract behavior modeling language based on concurrent objects — and is based on the use of cost relations.

1.3 Outline

The rest of this report is organized as follows: Chapter 2 gives some necessary definitions and background, in particular it defines a simple probabilistic programming language and the formal meaning of expected cost; Chapter 3 is the core of this work, it describes all pieces required to infer upper-bounds on the expected cost of probabilistic programs; Chapter 4 describes a corresponding implementation; and Chapter 5 concludes, and discusses future and related work.

f

CHAPTER 2

Preliminaries

In this chapter we give some necessary definitions and background, in particular: overview of basic concepts in probability theory, definition of a simple probabilistic programming language that will be used throughout this work, definition of a corresponding operational semantics, definition of the different notions of cost that we are interested in and definition of the meaning of upper-bound and lower-bound cost functions.

2.1 Mathematical background

A *random variable* X is a variable whose value is determined by a random event, i.e., it is the outcome of a probabilistic experiment. It is *discrete* if it takes a finite or countable number of values, and *continuous* otherwise. In this work, we use only finite discrete random variables, and thus, from now on, a random variable stands for a finite discrete random variable, unless we explicitly state otherwise. If $v_1, \ldots, v_k$ are the possible values of X, we let $P(X = v_i)$ be the probability that X takes value v_i. It must satisfy the following conditions:

- $\forall i \in [1..k].\ 0 \leq P(X = v_i) \leq 1$; and
- $\sum_{i=1}^{k} P(X = v_i) = 1$

The *probability distribution* of a random variable is a list of the probabilities associated with each possible value. We often write it as $\{v_1 : p_1, \ldots, v_i : p_i\}$ where $p_i = P(X = v_i)$. We will mainly use a uniform distribution which assigns equal probability $\frac{1}{k}$ to all values $v_1, \ldots, v_k$. The expected value of a random variable X is defined as

$$E[X] = \sum_{i=1}^{k} P(X = v_i) \cdot v_i$$

In what follows $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ are used to denote the set of integer, rational and real numbers respectively. Moreover, $\mathbb{Z}_{\geq 0}$, $\mathbb{Q}_{\geq 0}$ and $\mathbb{R}_{\geq 0}$ denote the corresponding sets of nonnegative numbers.

2.2 Probabilistic Programs

In this section, we define the syntax and semantics of the probabilistic programs that we will use in this work.

2.2.1 Syntax

A probabilistic program (or simply a program) P is a sequence of instructions that adhere to the following grammar, i.e., it is constructed starting from the grammar symbol c:

$$\begin{aligned}
aop &\equiv\ + \mid - \mid * \mid / \\
bop &\equiv\ > \mid < \mid \leq \mid \geq \mid == \mid !\!= \\
e &\equiv\ \mathsf{id} \mid n \mid e_1\ aop\ e_2 \\
b &\equiv\ \textbf{false} \mid \textbf{true} \mid e_1\ bop\ e_2 \mid b_1\ \textsf{and}\ b_2 \mid b_1\ \textsf{or}\ b_2 \mid \textsf{not}\ b \\
c &\equiv\ \textbf{skip} \mid \textbf{tick}(\mathsf{ce}) \mid \mathsf{id}\ :=\ e \mid \mathsf{id}\ :=\ e + R_\mu \mid c_1\ \oplus_p\ c_2 \mid c_1\ \diamond\ c_2 \\
&\quad\ \textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \mid \textbf{while}\ b\ s \mid c_1; c_2
\end{aligned}$$

where

- The definitions of arithmetic and Boolean expressions e and b, respectively, are quite common: id is an integer program variable, n is an integer number, *aop* is an arithmetic operator, and *bop* is a Boolean comparison operator.

- Instruction "**skip**" does nothing.

- Instruction "**tick**(ce)" is used to model resource consumption, i.e., executing it costs ce units where ce is an arithmetic expression (the exact form is given in Definition 2.2.1). It is the only instruction that consumes resources.

- Instruction "$\mathsf{id} := e$" evaluates expression e and assigns the result to variable id.

- Instruction "$\mathsf{id} := e + R_\mu$" is a probabilistic assignment, where R is a random variable whose probability distribution is μ. Its execution independently samples a value v for R, and then evaluates $e + v$ and assigns the result to id. As mentioned above, we view μ as a set of k pairs of the form $v_i : p_i$ indicating that the probability of sampling v_i is p_i. We will mainly use uniform distribution in our examples, and thus, instead of R_μ we will write **unif**(S) where S is a set (or interval) of k values each one with probability $\frac{1}{k}$. For example, "$\mathsf{id} := e +$ **unif**(1..3) " takes the possible values from the set $S = \{1, 2, 3\}$. In general, when using **unif**($a..b$) we refer to the set $S = \{a, a + 1, \ldots, b\}$, requiring that $a < b$. Moreover, we drop e when it is 0.

- Instruction "$c_1 \diamond c_2$" is a nondeterministic branching, i.e., the execution proceeds either with c_1 or with c_2 in a nondeterministic way.

- Instruction "$c_1 \ \oplus_p \ c_2$" represents an independent probabilistic branching, i.e., with probability p the execution proceeds with c_1 and with probability $(1 - p)$ with c_2.

- Instruction "**if** b **then** c_1 **else** c_2" is a conditional statement, i.e., if the Boolean expression is evaluated to **true** the execution continues with c_1, otherwise it continues with c_2.

- Instruction "**while** b c" is while loop, i.e., c is executed as far as the guard b evaluates to **true**.

- Instruction "$c_1; c_2$" is a composition, and it is used to construct programs with several instructions.

The *ordered* set of all variables that appear in a given program P will be denoted by $\texttt{VARS}_P$ (or simply VARS when it is clear from the context). We assume that there are exactly n variables, all of type integer.

Next we define the syntax of cost expressions that can be used, among other things, in instruction **tick**.

Definition 2.2.1. A cost expression ce is a symbolic arithmetic expression that adhere to the following grammar

$$ce \equiv c \;\mid\; c \cdot \|l\| \;\mid\; ce_1 + ce_2$$

where $c \in \mathbb{Q}_{\geq 0}$ is a nonnegative rational number, and l is a linear expression of the form $a_0 + \sum_{i=1}^{m} a_i \cdot x_i$ such that $a_i \in \mathbb{Q}$ are rational numbers, $x_i \in \texttt{VARS}_P$ are program variables, and $\|l\| = \max(0, l)$.

2.2.2 Semantics

An *assignment* is a mapping $\sigma : \texttt{VARS}_P \mapsto \mathbb{Z}$ that maps program variables to integer values. We often write it as $(v_1, \ldots, v_n) \in \mathbb{Z}^n$ meaning that v_i is the value of the ith variable in $\texttt{VARS}_P$. Given an assignment σ, we use $\sigma[\mathsf{id} \mapsto v]$ to denote the new assignment obtained by changing the value of variable id to v in σ. A (program)

state s is a tuple of the form $\langle c \mid \sigma \rangle$, meaning that the current values of the variables are as defined by σ and that we still have to execute c.

A *transition* represents a single execution step, and is written as:

$$\langle c \mid \sigma \rangle \xrightarrow{(p,q)} \langle c' \mid \sigma' \rangle \tag{2.1}$$

Its meaning is as follows: from state $\langle c \mid \sigma \rangle$ we can move to state $\langle c' \mid \sigma' \rangle$, with probability p and consuming q resources, by executing the first instruction of c. Note that $0 \leq p \leq 1$ and $q \geq 0$ are rational numbers. When $p = 1$ and $q = 0$, we simply drop (p, q) from the transition. We allow c' to be the special symbol ϵ (empty sequence) as well to denote the end of an execution.

The operational semantics is depicted in Figure 2.1. It consists of rules defining valid transitions. Let us explain the different rules, in particular the ones related to probabilistic and nondeterministic instructions as the other rules are quite standard [44].

- Rule [`passign`] is for the probabilistic assignment "$\texttt{id} = e + R_\mu$". As we have explained before, its execution independently samples a value v_i from the corresponding distribution μ, evaluates $e + v_i$ into v, and assigns the result v to variable id. The new state is the result of updating the value of id in the previous state, and the resulting transition is annotated with the corresponding probability p_i.

- Rules $[\texttt{prob}]_1$ and $[\texttt{prob}]_2$ represent the two possible paths in the independent probabilistic choice "$c_1 \ \oplus_p \ c_2$". In both the transition is annotated with the corresponding probability, i.e., p or $1 - p$.

- Rules $[\texttt{nondet}]_1$ and $[\texttt{nondet}]_2$ represent the two possible paths in the nondeterministic choice "$c_1 \diamond c_2$". The probability is 1 and the cost is 0 in both cases, so the transition is not annotated.

$$[\texttt{skip}] \quad : \quad \langle \textsf{skip}; c \mid \sigma \rangle \to \langle c \mid \sigma \rangle$$

$$[\texttt{assign}] \quad : \quad \frac{v = [\![e]\!]_\sigma \,, \quad \sigma' = \sigma\,[\mathsf{id} \mapsto v]}{\langle \mathsf{id} = e; c \mid \sigma \rangle \to \langle c \mid \sigma' \rangle}$$

$$[\texttt{passign}] \quad : \quad \frac{v_i : p_i \in \mu, \quad v = [\![e]\!]_\sigma + v_i, \quad \sigma' = \sigma\,[\mathsf{id} \mapsto v]}{\langle \mathsf{id} = e + R_\mu; c \mid \sigma \rangle \xrightarrow{(p_i,0)} \langle c \mid \sigma' \rangle}$$

$$[\texttt{prob}]_1 \quad : \quad \langle c_1 \oplus_p c_2; c \mid \sigma \rangle \xrightarrow{(p,0)} \langle c_1; c \mid \sigma \rangle$$

$$[\texttt{prob}]_2 \quad : \quad \langle c_1 \oplus_p c_2; c \mid \sigma \rangle \xrightarrow{(1-p,0)} \langle c_2; c \mid \sigma \rangle$$

$$[\texttt{nondet}]_1 \quad : \quad \langle c_1 \diamond c_2; c \mid \sigma \rangle \to \langle c_1; c \mid \sigma \rangle$$

$$[\texttt{nondet}]_2 \quad : \quad \langle c_1 \diamond c_2; c \mid \sigma \rangle \to \langle c_2; c \mid \sigma \rangle$$

$$[\texttt{tick}] \quad : \quad \langle \textsf{tick}(ce); c \mid \sigma \rangle \xrightarrow{(1,[\![ce]\!]_\sigma)} \langle c \mid \sigma \rangle$$

$$[\texttt{if}]_{\texttt{true}} \quad : \quad \frac{[\![b]\!]_\sigma = \textsf{true}}{\langle \textsf{if} \;\; b \;\textsf{then}\; c_1 \;\textsf{else}\; c_2; c \mid \sigma \rangle \to \langle c_1; c \mid \sigma \rangle}$$

$$[\texttt{if}]_{\texttt{false}} \quad : \quad \frac{[\![b]\!]_\sigma = \textsf{false}}{\langle \textsf{if} \;\; b \;\textsf{then}\; c_1 \;\textsf{else}\; c_2; c \mid \sigma \rangle \to \langle c_2; c \mid \sigma \rangle}$$

$$[\texttt{while}]_{\texttt{true}} \quad : \quad \frac{[\![b]\!]_\sigma = \textsf{true}}{\langle \textsf{while}\; b\; c_1\; ; c \mid \sigma \rangle \to \langle c_1; \textsf{while}\; b\; c_1\; ; c \mid \sigma \rangle}$$

$$[\texttt{while}]_{\texttt{false}} \quad : \quad \frac{[\![b]\!]_\sigma = \textsf{false}}{\langle \textsf{while}\; b\; c_1\; ; c \mid \sigma \rangle \to \langle c \mid \sigma \rangle}$$

Figure 2.1: *Operational Semantics*

- Rule [tick] is for instruction **tick**(*ce*) that consumes *ce* resources. The transition is annotated with the result of evaluating *ce* in the current state σ, and with probability 1.

The semantic rules of the remaining instructions are quite standard, and thus we skip them.

An execution of a program P, starting from an initial assignment σ_0, can be described as a *trace* t of the form:

$$t \equiv s_0 = \langle P; \epsilon \mid \sigma_0 \rangle \xrightarrow{(p_0,q_0)} s_1 \xrightarrow{(p_1,q_1)} s_2 \xrightarrow{(p_2,q_2)} \cdots \tag{2.2}$$

where each $s_i \xrightarrow{(p_i,q_i)} s_{i+1}$ is a valid transition. A trace can be finite or infinite. Finite traces *must* end in a state of the form $\langle \epsilon \mid \sigma' \rangle$, i.e., they are complete executions. Note that the special symbol ϵ is concatenated to P in the initial state in order to identify the end of a finite execution. We use $\texttt{TRACES}(P, \sigma_0)$ to denote the set of all possible finite and infinite traces that start in the program state $\langle P; \epsilon \mid \sigma_0 \rangle$.

2.3 Expected cost of probabilistic programs

We start by discussing the classical notion of worst-case cost. For this, we ignore the probabilistic behavior of our programs and treat all probabilistic choices as nondeterministic. Afterwards, we formally define the notion of expected cost.

For a given trace t, we let $\texttt{CE}(t)$ and $\texttt{PR}(t)$ be the (ordered) sets of all corresponding cost and probability annotations, respectively. The cost of a trace t is defined as the sum of its resource annotations, formally:

$$\texttt{COST}(t) = \sum_{v \in \texttt{CE}(t)} v \tag{2.3}$$

The worst-cost of executing a program P with respect to an initial assignment σ_0 is the maximum cost of all possible traces, formally:

$$\texttt{COST}(P, \sigma_0) = \max\{\texttt{COST}(t) \mid \ t \in \texttt{TRACES}(P, \sigma_0)\} \tag{2.4}$$

Note that it is the worst-case cost for a given input, and not the worst-case cost among all inputs. The best-case cost can be defined similarly by changing max by min. In absence of nondeterministic instructions, the best-case and worst-case cost coincide for a given σ_0, while they do not when considering any possible input. The worst-case cost of P with respect to *all inputs* is a function $f : \mathbb{Z}^n \mapsto \mathbb{R}_{\geq 0} \cup \{\infty\}$ such that $f(\sigma) = \texttt{COST}(P, \sigma)$ for any initial assignment σ. A similar notion can be defined for best-case cost.

This notion of worst-case cost is not adequate for probabilistic programs, mainly because it does not take into account the likelihood of a trace, which is the product of its probability annotations and is formally defined as:

$$\mathbf{Pr}(t) = \prod_{p \in \texttt{PR}(t)} p \tag{2.5}$$

This leads us to the notion of expected cost, which takes this information into account.

Let us start with deterministic programs, i.e., assuming that we do not have the instruction $c_1 \diamond c_2$. In this case, the expected cost for an initial σ_0 is defined as:

$$\texttt{ECOST}(P, \sigma_0) = \sum_{t \in \texttt{TRACES}(P, \sigma_0)} \mathbf{Pr}(t) \cdot \texttt{COST}(t) \tag{2.6}$$

Namely, the contribution of every trace $t \in \texttt{TRACES}(P, \sigma_0)$ is the multiplication of its cost $\texttt{COST}(t)$ by its probability $\mathbf{Pr}(t)$. The expected cost for any input is a function $f : \mathbb{Z}^n \mapsto \mathbb{R}_{\geq 0} \cup \{\infty\}$ such that $f(\sigma) = \texttt{ECOST}(P, \sigma)$.

EXAMPLE 2.3.1. Consider again the program:

```
while ( n>0 ) {
  tick(1);
  n=n−1; ⊕3/4 skip;
}
```

For a given input n_0, infinite executions must take the first branch of $\oplus$ $j < n_0$ times and the second branch infinitely many. The probability of such a trace is $\lim_{i\to\infty}(\frac{3}{4})^j \cdot (\frac{1}{4})^i = 0$ and thus it contributes 0 to the expected cost.

A terminating execution starting from $n_0 \geq 1$ takes the first branch n_0 times and the second $i \geq 0$ times, i.e., the length of the trace is $n_0 + i$. The probability of such a trace is $\left(\frac{3}{4}\right)^{n_0} \cdot \left(\frac{1}{4}\right)^i$ and its cost is $n_0 + i$, and thus it contributes $\left(\frac{3}{4}\right)^{n_0} \cdot \left(\frac{1}{4}\right)^i \cdot (n_0 + i)$ to the expected cost. Let us compute the number of possible traces of length $n_0 + i$. Since the last step of such a trace takes the first branch, the remaining $n_0 - 1$ times of taking the first branch can be placed anywhere in the first $n_0 + i - 1$ steps, and thus the total number of traces of length $n_0 + i$ is $\binom{n_0+i-1}{n_0-1}$. Now the expected cost for $n_0 \geq 1$ is:

$$\sum_{i=0}^{\infty} \binom{n_0 + i - 1}{n_0 - 1} \cdot \left(\frac{3}{4}\right)^{n_0} \cdot \left(\frac{1}{4}\right)^i \cdot (n_0 + i) = \frac{4}{3} \cdot n_0$$

For $n_0 \leq 0$ the expected cost is 0 since the loop is not executed. Therefore, the expected cost of the program is $\texttt{ECOST}(P, (n_0)) = \frac{4}{3} \cdot \|n_0\|$. □

In the presence of nondeterminism, the definition of expected cost is more elaborated, and has been considered before in [28]. Intuitively, we can think of it as follows: suppose that we are given a *scheduler* that decides (in a deterministic way) which choices to make for the nondeterministic instructions. Using this scheduler would make our programs deterministic and thus we can use the expected cost as in Equation (2.6), which is valid only for this particular scheduler. Now, the expected cost for nondeterministic programs is the maximum among the expected costs obtained by using all possible schedulers. This notion is formalized in [28] using Markov Decision Processes. The fundamental contribution of [28], however, is a definition of the expected cost that relies on a *weakest precondition* calculus, which makes reasoning on the expected cost simpler. We rely on this calculus, mainly as presented in [33].

The weakest precondition calculus is depicted in Figure 2.2. The meaning of

c	$\mathbf{ert}[c](f)$
skip	f
$\mathsf{id} = e$	$f[\mathsf{id} \,/\, [\![e]\!]_\sigma]$
$\mathsf{id} = e + R_\mu$	$\Sigma_{p:v\in\mu}\ p \cdot f[\mathsf{id} \,/\, ([\![e]\!]_\sigma + v)]$
$c_1 \ \diamond \ c_2$	$\max(\mathbf{ert}[c_1](f), \mathbf{ert}[c_2](f))$
$c_1 \ \oplus_p \ c_2$	$p \cdot \mathbf{ert}[c_1](f) + (1-p) \cdot \mathbf{ert}[c_2](f)$
$\mathsf{tick}(ce)$	$f + \lambda\sigma.\, [\![ce]\!]_\sigma$
$\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$	$\mathcal{B}_\delta(b) \cdot \mathbf{ert}[c_1](f) + \mathcal{B}_\delta(\neg b) \cdot \mathbf{ert}[c_2](f)$
$\mathsf{while}\ b\ c$	$\mathsf{lfp}\ (F)$ where $F = \lambda X. \mathcal{B}_\delta(b) \cdot \mathbf{ert}[c](X) + \mathcal{B}_\delta(\neg b) \cdot f$
$c_1; c_2$	$\mathbf{ert}[c_1](\mathbf{ert}[c_2](f))$

Figure 2.2: $\max(f_1, f_2)$ *should be interpreted as* $\lambda\sigma.\max\{f_1(\sigma), f_2(\sigma)\}$ *and* $f_1 + f_2$ *as* $\lambda\sigma.\,(f_1(\sigma) + f_2(\sigma))$. *lfp denotes the least fix-point and* $\mathcal{B}_\delta(b)$ *denotes the function that returns* 1 *if* b *is* ***true*** *and* 0 *otherwise.*

$\mathbf{ert}[c](f)$ is as follows: assuming that the expected cost of what comes after c is modeled by function $f : \mathbb{Z}^n \mapsto \mathbb{R}_{\geq 0} \cup \{\infty\}$, then $\mathbf{ert}[c](f)$ is a function (from $\mathbb{Z}^n$ to $\mathbb{R}_{\geq 0} \cup \{\infty\}$) that models the expected cost of c and what comes after c. We refer to f as the expected cost of the *continuation*. Let us explain some of the rules:

- The case of **skip** is trivial, it takes the expected cost of the continuation f.
- The case of assignment replaces id by the expression e in the continuation f, which is equivalent to saying that it returns $\lambda\sigma.f(\sigma[\mathsf{id} \,/\, [\![e]\!]_\sigma])$.
- The case of probabilistic assignment takes the weighted sum of the expected cost of the continuation f for all $v : p \in \mu$.
- The case of nondeterministic choice takes the maximum value of the expected cost of both branches.
- The case of probabilistic choice takes the weighted sum of the expected cost of both branches.
- The case of **tick** adds the value ce to the expected cost of the continuation.

- The case of **if** takes either the expected cost of the then or the else branch, depending on the value of b. Here $\mathcal{B}_\delta(b)$ return 1 if b evaluates to **true** and 0 otherwise.

- The case of the **while** accounts for executing the body arbitrarily number of iterations, and then the continuation, using a least fixpoint of a corresponding operator F.

- The case of sequential composition constructs first a function for the expected cost of c_2, and then uses it as continuation when constructing that of c_1.

Using **ert**, the expected cost of a program P is defined as follows.

Definition 2.3.2. The expected cost of a program P is defined by the function $\mathbf{ert}[P](\mathbf{0})$, where $\mathbf{0} \equiv \lambda\sigma.0$.

Since $\mathbf{ert}[P](\mathbf{0})$ is not necessarily computable, in this work we are interested in automatically inferring a closed-form upper-bound function $f : \mathbb{Z}^n \mapsto \mathbb{R}_{\geq 0} \cup \{\infty\}$ on the expected cost.

Definition 2.3.3. We say that function $f : \mathbb{Z}^n \mapsto \mathbb{R}_{\geq 0} \cup \{\infty\}$ is an upper-bound on $\mathbf{ert}[P](\mathbf{0})$ if for any $\sigma \in \mathbb{Z}^n$ we have $f(\sigma) \geq \mathbf{ert}[P](\mathbf{0})(\sigma)$.

CHAPTER 3

Inference of Expected Cost via Cost Relations

Recurrence equations are a classical mechanism that is used to model the cost of programs, and they have been used for inferring closed-form upper-bound (resp. lower-bound) functions on the *worst-case* (resp. best-case) cost [43, 25, 41, 34, 17, 23, 23, 9, 40]. The main advantage of using recurrence equations is the availability of automatic tools for solving them into closed-form functions, such as `PURRs` [8] and `Mathematica` [30]. However, in spite of their popularity, due to their deterministic nature recurrence equations cannot easily model the cost of nondeterministic programs. Yet another important limitation, is that solving recurrence equations with multiple variables is significantly more difficult than solving recurrence equations with one variable, and in some cases, it is not even feasible.

The limitations just discussed have a significant impact in practice, since tools that rely on recurrence equations would not be able to handle a wide class of programs. Even if the programming language under consideration is deterministic, static analyzers typically need to introduce nondeterminism (in the intermediate language) in order to handle a wide range of programs. For example, in termination and cost analysis, a common approach to handling programs that manipulate data structures is

to abstract them using nondeterministic linear constraints that model their sizes [39]. It is needless to say that the cost of programs often depends on multiple variables.

To overcome the above limitations, recurrence equations were generalized to what is called *cost relations* [3], which are very similar to recurrence equations but that allow some degree of nondeterminism. Several techniques for solving cost relations into closed-form functions, that represent upper-bounds (resp. lower-bounds) on the worst-case (resp. best-case) cost, have been developed [3, 5, 6, 21, 20]. These tools perform well in practice, both in terms of performance and precision, even in the presence of nontrivial nondeterminism and multiple variables.

In what follows we show that the above considerations are also valid when considering the *expected cost* of probabilistic programs: (1) we show how the expected cost of probabilistic programs can be modeled with recurrence equations; (2) we explore the limitations of recurrence equations in this context and suggest the use of an *extended form of cost relations* in order to handle nondeterminism; (3) we describe how probabilistic programs can be automatically transformed into cost relations that model their expected cost; and (4) finally we describe how to solve cost relations into closed-form upper-bound functions on the expected cost.

3.1 Modeling the expected cost with cost relations

In this section, we informally describe how to model the expected cost of probabilistic programs with cost relations. First, we start with a simple example whose expected cost can be modeled with recurrence equations.

EXAMPLE 3.1.1. Consider again the following program:

```
while (n>0) {
  tick(1);
  n := n−1; ⊕_{3/4} skip;
}
```

Note that it has a probabilistic branching instruction that decrements variable n in one branch, and leaves it without a change in the other.

The expected cost of this program for a given input n, denoted by $C(n)$, can be modeled using the following recurrence equations:

$$\begin{array}{lll} C(n) = & 0 & n = 0 \\ C(n) = & 1 + \frac{3}{4} \cdot C(n-1) + \frac{1}{4} \cdot C(n) & n > 0 \end{array}$$

Here the first equation models the base-case, i.e., when n is 0, and the second models the case in which n is positive, i.e., when entering the loop. It is easy to see that the second equation coincides with the definition of cases **tick** and $c_1 \oplus_p c_2$ in the **ert** calculus of Figure 2.2. This cost relation can be solved using an off-the-shelf solver into a closed-form function $C(n) = \frac{4}{3} \cdot n$ which is the *exact* expected cost of this program. Then, in order to handle the case in which n is negative, we use $C(n) = \frac{4}{3} \cdot \|n\|$. □

Not all programs can be modeled with standard recurrence equations as in the above example, this is mainly because of nondeterminism that is either explicit in the language, as the nondeterministic choice of our language, or implicit due to abstractions that are typically applied by static analysis.

EXAMPLE 3.1.2. Consider the following program:

```
while (n>0 and m>0) {
  tick(1);
  (n := n−1 ⋄ m := m−1) ⊕_{3/4} skip;
}
```

Note that in one branch of the probabilistic choice we nondeterministically decrease either n or m by 1, and in the other branch they are not modified. Let us translate it into a set of equations that models its expected cost, as we have done in Example 3.1.1:

$$\begin{array}{lll} C(n,m) = & 0 & n \leq 0 \\ C(n,m) = & 0 & m \leq 0 \\ C(n,m) = & 1 + \frac{3}{4} \cdot C(n-1,m) + \frac{1}{4} \cdot C_3(n,m) & n > 0, m > 0 \\ C(n,m) = & 1 + \frac{3}{4} \cdot C(n,m-1) + \frac{1}{4} \cdot C_3(n,m) & n > 0, m > 0 \end{array}$$

The first two equations handle the base-case of the while loop and the other equations handle the $\mathsf{tick}(1)$ instruction together with the probabilistic choice. These equations are not valid recurrence equations, mainly due to the nondeterminism in the applicability conditions (the constraints on the right), i.e, for the same values of n and m, we can apply one equation or another. These equations actually form a cost relation [3] as we will see shortly, but in some extended form to allow multiplying calls by a constant, e.g., $\frac{1}{4} \cdot C(n, m)$. Solving this cost relation into a closed-form upper-bound function is beyond the capabilities of existing techniques, mainly because of calls such as $\frac{1}{4} \cdot C(n, m)$. □

Developing techniques for solving cost relations of this extended form is the main goal of this work. In what follows, in sections 3.2 and 3.3 we define the notion of cost relations and describe how to automatically generate cost relations from probabilistic programs that model their expected cost, and in sections 3.4 and 3.5 we propose a technique for automatically solving such cost relations into closed-form upper-bound functions on the expected cost.

3.2 From probabilistic programs to cost relations

Let us first fix some notation. We use $\bar{x}$ (possibly with a subscript) to denote a sequence of variables $x_1, \ldots, x_n$. We use ϕ to denote a constraint of the form e_1 *bop* e_2 (as the conditions of our language – see Section 2.2). A conjunction $\phi_1 \wedge \cdots \wedge \phi_k$ is denoted by φ, and it is often written as $[\phi_1, \ldots, \phi_k]$. The empty list of constraints [ ] represents the empty conjunction, which corresponds to *true*.

Definition 3.2.1. A cost relation $\mathcal{R}$ is a set of equations of the form:

$$\langle C(\bar{x}) = ce + \sum_{i=1}^{m} p_i \cdot C_i(\bar{x}_i), \varphi \rangle$$

where:

1. φ is a list of constraints over variables $\bar{x}, \bar{x}_1, \ldots, \bar{x}_m$, and probably some other existentially quantified (i.e., local) variables.

2. ce is a cost expression as in Definition 2.2.1, and is called the local cost.

3. Each p_i is a positive rational number.

4. C and C_i are called cost relation symbols, and each must be defined by some equation in $\mathcal{R}$, i.e., appears in the left-hand side of some equation of $\mathcal{R}$.

The set of all cost relation symbols in $\mathcal{R}$ is denoted by $\texttt{crsym}(\mathcal{R})$. □

When writing cost equations, for simplicity, and when it does not create any confusion: we drop the enclosing angle brackets $\langle\ \rangle$; we drop φ when it is [ ] (i.e., *true*); and we write arithmetic expressions directly as parameters, e.g., $C(n-1, m)$.

EXAMPLE 3.2.2. Consider again the equations of Example 3.1.2, and note that they form a cost relation that consists of 4 equations and one cost relation symbol C, i.e., $\texttt{crsym}(\mathcal{R}) = \{C\}$. □

Next, we explain how to automatically transform a given probabilistic program P into a cost relation $\mathcal{R}_P$ that models its expected cost.

Let us set some notation first:

- We use $\bar{x}$ for the tuple of all variables in the programs. As the reader might have noticed, a program variable `n` is written in a different font as n in the corresponding cost relation – we use both forms to refer to the variable `n`.

- We use $\bar{x}[id_1/id_2]$ to denote the sequence resulting from $\bar{x}$ by replacing variable id_1 by id_2.

- Given a Boolean condition b (as the conditions of our language – see Section 2.2), we use $\texttt{DNF}(b)$ to obtain the corresponding `DNF` representation as a set of lists

of constraints. For example, applying $\texttt{DNF}((x > y \text{ or } y < z) \text{ and } x > 2w)$ results in $(x > y \text{ and } x > 2w) \text{ or } (y < z \text{ and } x > 2w)$ that we write as $\{[x > y, x > 2w], [y < z, x > 2w]\}$ — each list represents a conjunction of its elements, and the set represents a disjunction of all these conjunctions.

- We assume that each instruction is annotated with a unique label ℓ. The label of a sequence of instructions is defined as the label of its first instruction, this means that if c is a sequence of instructions, and we write c^ℓ, then ℓ coincides with the label of the first instruction in c.

The transformation of a program P into a cost relation $\mathcal{R}_P$ is depicted in Figure 3.1. The first argument of $\mathcal{T}$ is a program, and the second is a cost relation symbol representing the expected cost of what comes after the program. Intuitively, for each instruction with label ℓ, it generates a set of equations defining $C_\ell(\bar{x})$ that model the expected cost when starting the execution from the instruction with label ℓ. We first define how the cost relation $\mathcal{R}_P$ is obtained and then we explain the details of the transformations.

Definition 3.2.3. Given a program P, its expected cost relation $\mathcal{R}_P$ is

$$\mathcal{R}_P = \mathcal{T}(P, C_\bullet) \cup \{\langle C_\bullet(\bar{x}) = 0, [\,]\rangle\}$$

where $C_\bullet$ is a special cost relation symbol representing 0 cost. □

Let us explain how $\mathcal{T}$ generates cost equations for each instruction. The call $\mathcal{T}(c, C)$ generates cost equations that model the expected cost of instruction c (which might be a sequence of instructions), assuming that the expected cost of what comes after c is modeled by C (the continuation). We will see later how the cost equations of c call C to account for the cost of the code that comes after c. The call $\mathcal{T}(P, C_\bullet)$ generates cost equations for program P assuming that the expected cost of what

$\mathcal{T}(P, C)$ {

case P {

skip^ℓ	$\Rightarrow$	$\{\langle C_\ell(\bar{x}) = C(\bar{x}), [\,]\rangle\}$
$\mathsf{id} :=^\ell e$	$\Rightarrow$	$\{\langle C_\ell(\bar{x}) = C(\bar{x}[id/id']), [id' = e]\rangle\}$
$\mathsf{id} :=^\ell e + R_\mu$	$\Rightarrow$	$\{\langle C_\ell(\bar{x}) = \sum_{p_i : v_i \in \mu} p_i \cdot C(\bar{x}[id/id'_i]), \varphi\rangle\}$ where $\varphi = [id'_1 = e + v_1, \ldots, id'_k = e + v_k]$
$c_1^{\ell_1}; \ldots; c_m^{\ell_m}$	$\Rightarrow$	$\mathcal{T}(c_m^{\ell_m}, C) \cup \mathcal{T}(c_{m-1}^{\ell_{m-1}}, C_{\ell_m}) \cup \cdots \cup \mathcal{T}(c_1^{\ell_1}, C_2)$
$c_1^{\ell_1} \diamond^\ell c_2^{\ell_2}$	$\Rightarrow$	$\mathcal{T}(c_1^{\ell_1}, C) \cup \mathcal{T}(c_2^{\ell_2}, C) \cup$ $\{\langle C_\ell(\bar{x}) = C_{\ell_1}(\bar{x}), [\,]\rangle, \langle C_\ell(\bar{x}) = C_{\ell_2}(\bar{x}), [\,]\rangle\}$
$c_1^{\ell_1} \oplus_p^\ell c_2^{\ell_2}$	$\Rightarrow$	$\mathcal{T}(c_1^{\ell_1}, C) \cup \mathcal{T}(c_2^{\ell_2}, C) \cup$ $\{\langle C_\ell(\bar{x}) = p \cdot C_{\ell_1}(\bar{x}) + (1-p) \cdot C_{\ell_2}(\bar{x}), [\,]\rangle\}$
$\mathsf{tick}(ce)^\ell$	$\Rightarrow$	$\{\langle C_\ell(\bar{x}) = ce + C(\bar{x}), [\,]\rangle$
if b^ℓ **then** $c_1^{\ell_1}$ **else** $c_2^{\ell_2}$	$\Rightarrow$	$\mathcal{T}(c_1^{\ell_1}, C) \cup \mathcal{T}(c_2^{\ell_2}, C) \cup$ $\{\langle C_\ell(\bar{x}) = C_{\ell_1}(\bar{x}), \varphi\rangle \mid \varphi \in \mathtt{DNF}(b)\} \cup$ $\{\langle C_\ell(\bar{x}) = C_{\ell_2}(\bar{x}), \varphi\rangle \mid \varphi \in \mathtt{DNF}(not\ b)\}$
while b^ℓ c^{ℓ_1}	$\Rightarrow$	$\mathcal{T}(c^{\ell_1}, C_\ell) \cup$ $\{\langle C_\ell(\bar{x}) = C_{\ell_1}(\bar{x}), \varphi\rangle \mid \varphi \in \mathtt{DNF}(b)\} \cup$ $\langle C_\ell(\bar{x}) = C(\bar{x}), \varphi\rangle \mid \varphi \in \mathtt{DNF}(not\ b)\}$

}

}

Figure 3.1: *From probabilistic programs to cost relations*

comes after P is modeled by $C_\bullet$, i.e., 0. Note that this is very similar to the definition of expected cost as given in Definition 2.3.2 – here $C_\bullet$ plays the role of $\mathbf{0}$.

Let us now explain all cases of $\mathcal{T}$:

- skip^ℓ: since its expected cost is 0, we generate the single equation $C_\ell(\bar{x}) = C(\bar{x})$ which states that the expected cost is as that of the continuation C. Note that ℓ is the label of the instruction.

- $\mathsf{id} :=^\ell e$: we generate an equation stating that the expected cost is as that of the continuation C, but taking into account the change of the value of id to e. This is done by

  1. Introducing a new variable id' that represents the new value of id.
  2. Adding the constraint $[id' = e]$.
  3. Using the sequence of variables $\bar{x}[id/id']$ in the call to C, which stands for replacing id by id' in $\bar{x}$.

- $\mathsf{id} :=^\ell e + R_\mu$: assuming that the probability distribution μ consists of the elements $\{p_1 : v_1, \ldots, p_k : v_k\}$, we generate an equation stating that the expected cost is the sum of $p_i \cdot C(\bar{x}[id/id'_i])$, and add $[id'_1 = e + v_1, \ldots, id'_k = e + v_k]$ as constraints.

- $c_1^{\ell_1}; \ldots; c_m^{\ell_m}$: for each c_i we recursively call $\mathcal{T}$ to generate its corresponding equations. The difference between these calls is the continuation: for c_n the continuation is C, because it is the expected cost of what comes after the composition, while for any other c_i it is $C_{\ell_{i+1}}$, i.e., the expected cost of what comes after c_i.

- $c_1^{\ell_1} \diamond^\ell c_2^{\ell_2}$: we recursively call $\mathcal{T}$ to generate the corresponding cost equations for each c_i, in both cases, C is used as a continuation. In addition, we generate two

equations for C_ℓ that simulate the nondeterministic choice, one calls C_{ℓ_1} and the other calls C_{ℓ_2}.

- $c_1^{\ell_1} \oplus_p^{\ell} c_2^{\ell_2}$: we recursively call $\mathcal{T}$ to generate the corresponding cost equations for each c_i, in both cases, C is used as a continuation. In addition, we generate an equation for C_ℓ that accounts for the probabilistic choice, i.e., defines the expected cost as p multiplied by the expected cost C_{ℓ_1} of the first branch, plus $(1-p)$ multiplied by the expected cost C_{ℓ_2} of the second branch.

- $\mathsf{tick}(ce)^{\ell}$: this is the only instruction that consumes resources, so we generate an equation stating that its expected cost is ce plus the expected cost of the continuation C.

- $\mathsf{if}\ \ b^{\ell}\ \mathsf{then}\ c_1^{\ell_1}\ \mathsf{else}\ c_2^{\ell_2}$: we recursively call $\mathcal{T}$ to generate the corresponding cost equations for each c_i, in both cases, C is used as a continuation. For each conjunction $\varphi \in \texttt{DNF}(b)$ (resp. $\varphi \in \texttt{DNF}(\mathit{not}\ b)$) we generate a corresponding equation for C_ℓ that calls C_{ℓ_1} of c_1 (resp. C_{ℓ_2} of c_2).

- $\mathsf{while}\ b^{\ell}\ c^{\ell_1}$: we recursively call $\mathcal{T}$ to generate the corresponding cost equations for c, with the continuation as C_ℓ since what comes after c is the while loop again (rest of iterations). Then for every conjunction $\varphi \in \texttt{DNF}(b)$ (resp. $\varphi \in \texttt{DNF}(\mathit{not}\ b)$), we generate a corresponding equation that calls C_{ℓ_1} of c (resp. C).

Note that each label in the program (i.e., each program point) has corresponding cost equations. In what follows we assume that the label of the first instruction in the program is ℓ_0, i.e., the equation of C_{ℓ_0} defines the expected cost of the program.

Programs sometimes come with invariants for each program point (i.e., label). These are constraints on the program variables that always hold when the execution reaches the corresponding program point. They are useful since they reveal some relations between variables that are not explicit in the program. Inferring invariants

for our probabilistic programs can be done using off-the-shelf invariant generators for non-probabilistic programs, simply by considering all probabilistic instructions as nondeterministic choices. Invariants can be incorporated in $\mathcal{R}_P$ as follows: the invariant of the program point corresponding to label ℓ is added to the constraints of all equations defining C_ℓ. This can have a crucial impact on precision as we will see later on this chapter.

Let us demonstrate how the transformation works on some examples.

EXAMPLE 3.2.4. Consider again the program of Example 3.1.1, together with labels for the different instructions:

```
while (n>0)^ℓ0 {
  tick(1)^ℓ1;
  n :=^ℓ3 n−1; ⊕_{3/4}^ℓ2 skip^ℓ4;
}
```

Applying $\mathcal{T}(P, C_\bullet)$, and adding the definition of $C_\bullet$, results in the following cost relation $\mathcal{R}_P$:

$$\begin{array}{rcll} C_{\ell_0}(n) = & C_\bullet(n) & [n \leq 0] \\ C_{\ell_0}(n) = & C_{\ell_1}(n) & [n > 0] \\ C_{\ell_1}(n) = & 1 + C_{\ell_2}(n) & [\mathbf{n > 0}] \\ C_{\ell_2}(n) = & \frac{3}{4} \cdot C_{\ell_3}(n) + \frac{1}{4} \cdot C_{\ell_4}(n) & [\mathbf{n > 0}] \\ C_{\ell_3}(n) = & C_{\ell_0}(n') & [n' = n - 1, \mathbf{n > 0}] \\ C_{\ell_4}(n) = & C_{\ell_0}(n) & [\mathbf{n > 0}] \\ C_\bullet(n) = & 0 & [\,] \end{array}$$

The constraints in bold font come from corresponding invariants. We can observe that if instruction c^{ℓ_j} is executed immediately after c^{ℓ_i}, then the expected cost C_{ℓ_i} is defined in terms of the expected cost C_{ℓ_j}. Observe that C_{ℓ_0}, which corresponds to the while loop, calls $C_\bullet$ in the base-case equation and calls C_{ℓ_1} in the other equation. This cost relation is not as compact as the one we manually generated in Example 3.1.1, shortly we will see how to simplify cost relations into such a compact form.

EXAMPLE 3.2.5. Consider again the program of Example 3.1.2, together with labels for the different instructions:

```
while (n>0 and m>0)^ℓ0 {
  tick(1)^ℓ1;
  (n :=^ℓ5 n−1 ⋄^ℓ3 m :=^ℓ6 m−1) ⊕^ℓ2_{3/4} skip^ℓ4;
}
```

The sequence of program variables $\bar{x}$ is (n, m). Applying $\mathcal{T}(P, C_\bullet)$, and adding the definition of $C_\bullet$ results in the following cost relation $\mathcal{R}_P$:

$$
\begin{array}{rcll}
C_{\ell_0}(n,m) = & C_\bullet(n,m) & [n \leq 0] \\
C_{\ell_0}(n,m) = & C_\bullet(n,m) & [m \leq 0] \\
C_{\ell_0}(n,m) = & C_{\ell_1}(n,m) & [n > 0, m > 0] \\
C_{\ell_1}(n,m) = & 1 + C_{\ell_2}(n,m) & [\mathbf{n > 0}, \mathbf{m > 0}] \\
C_{\ell_2}(n,m) = & \frac{3}{4} \cdot C_{\ell_3}(n,m) + \frac{1}{4} \cdot C_{\ell_4}(n,m) & [\mathbf{n > 0}, \mathbf{m > 0}] \\
C_{\ell_3}(n,m) = & C_{\ell_5}(n,m) & [\mathbf{n > 0}, \mathbf{m > 0}] \\
C_{\ell_3}(n,m) = & C_{\ell_6}(n,m) & [\mathbf{n > 0}, \mathbf{m > 0}] \\
C_{\ell_4}(n,m) = & C_{\ell_0}(n,m) & [\mathbf{n > 0}, \mathbf{m > 0}] \\
C_{\ell_5}(n,m) = & C_{\ell_0}(n',m) & [n' = n-1, \mathbf{n > 0}, \mathbf{m > 0}] \\
C_{\ell_6}(n,m) = & C_{\ell_0}(n,m') & [m' = m-1, \mathbf{n > 0}, \mathbf{m > 0}] \\
C_\bullet(n,m) = & 0 & [\,]
\end{array}
$$

Observe that C_{ℓ_0}, which corresponds to the while loop, calls $C_\bullet$ in the base-case equations and calls C_{ℓ_1} in the other equation. In addition, C_{ℓ_0} has two base-case equations because $\texttt{DNF}(\textsf{not}(n > 0 \textsf{ and } m > 0)) = \{[n \leq 0], [m \leq 0]\}$. The constraints in bold font correspond to invariants. □

EXAMPLE 3.2.6. Consider the following program:

```
while (x+3 <= n)^ℓ0 {
  if ( y<m )^ℓ1 then
    skip^ℓ5; ⊕^ℓ3_{1/2} y :=^ℓ6 y+1;
  else
    skip^ℓ7; ⊕^ℓ4_{1/4} x :=^ℓ8 x+unif(1..3)
  tick(1)^ℓ2;
}
```

The sequence of program variables $\bar{x}$ is (x, y, n, m). Applying $\mathcal{T}(P, C_\bullet)$, and adding

the definition of $C_\bullet$, results in the following cost relation $\mathcal{R}_P$:

$$
\begin{array}{rll}
C_{\ell_0}(x,y,n,m) = & C_\bullet(x,y,n,m), & [x+3>n] \\
C_{\ell_0}(x,y,n,m) = & C_{\ell_1}(x,y,n,m), & [x+3\leq n] \\
C_{\ell_1}(x,y,n,m) = & C_{\ell_3}(x,y,n,m) & [y<m, \mathbf{x}+\mathbf{3}\leq \mathbf{n}] \\
C_{\ell_1}(x,y,n,m) = & C_{\ell_4}(x,y,n,m) & [y\geq m, \mathbf{x}+\mathbf{3}\leq \mathbf{n}] \\
C_{\ell_2}(x,y,n,m) = & 1+C_{\ell_0}(x,y,n,m) & [\mathbf{x}+\mathbf{3}\leq \mathbf{n}] \\
C_{\ell_3}(x,y,n,m) = & \frac{1}{2}\cdot C_{\ell_5}(x,y,n,m)+ & [\mathbf{x}+\mathbf{3}\leq \mathbf{n}, \mathbf{y}<\mathbf{m}] \\
& \frac{1}{2}\cdot C_{\ell_6}(x,y,n,m) & \\
C_{\ell_4}(x,y,n,m) = & \frac{1}{4}\cdot C_{\ell_7}(x,y,n,m)+ & [\mathbf{x}+\mathbf{3}\leq \mathbf{n}, \mathbf{y}\geq \mathbf{m}] \\
& \frac{3}{4}\cdot C_{\ell_8}(x,y,n,m) & \\
C_{\ell_5}(x,y,n,m) = & C_{\ell_2}(x,y,n,m) & [\mathbf{x}+\mathbf{3}\leq \mathbf{n}, \mathbf{y}<\mathbf{m}] \\
C_{\ell_6}(x,y,n,m) = & C_{\ell_2}(x,y',n,m) & [y'=y+1, \mathbf{x}+\mathbf{3}\leq \mathbf{n}, \mathbf{y}<\mathbf{m}] \\
C_{\ell_7}(x,y,n,m) = & C_{\ell_2}(x,y,n,m) & [\mathbf{x}+\mathbf{3}\leq \mathbf{n}, \mathbf{y}\geq \mathbf{m}] \\
C_{\ell_8}(x,y,n,m) = & \sum_{i=1}^{3}\frac{1}{3}\cdot C_{\ell_2}(x'_i,y,n,m) & [x'_i=x+i, \mathbf{x}+\mathbf{3}\leq \mathbf{n}, \mathbf{y}\geq \mathbf{m}] \\
C_\bullet(x,y,n,m) = & 0 & [\,]
\end{array}
$$

Note that the expected cost of each instruction is defined in terms of the expected cost of its successors. □

Let us explain how cost relations model the expected cost of the corresponding program. Intuitively, if we are given a function $f_C : \mathbb{Z}^n \mapsto \mathbb{R}_{\geq 0} \cup \{\infty\}$ for each $C \in \texttt{crsym}(\mathcal{R}_P)$, then they form a valid upper-bound on the expected cost if for each equation $\langle C(\bar{x}) = ce + \sum_{i=1}^{m} p_i \cdot C_i(\bar{x}_i), \varphi\rangle \in \mathcal{R}_P$, the following formula holds

$$\varphi \models f_C(\bar{x}) \geq ce + \sum_{i=1}^{m} p_i \cdot f_{C_i}(\bar{x}_i) \tag{3.1}$$

Namely, $f_C(\bar{x})$ is enough to pay for its local resource consumption ce plus the expected cost of each $p_i \cdot f_{C_i}(\bar{x}_i)$. The symbol $\models$ means a logical implication, i.e., for any assignment that satisfies the constraints φ, the right-hand side is *true*. In what follows we refer to a formula like (3.1) as *cost equation formula*.

The following theorem summarizes this intuition. Recall that C_{ℓ_0} is the cost equation that corresponds to the first instruction in the program.

THEOREM 3.2.7. *Let P be a program, $\mathcal{R}_P$ its corresponding expected cost relation obtained as in Definition 3.2.3, and $F_{\mathcal{R}_P} = \{f_C : \mathbb{Z}^n \mapsto \mathbb{R}_{\geq 0} \cup \{\infty\} \mid C \in \texttt{crsym}(\mathcal{R}_P)\}$*

be a set of corresponding functions. If for each $\langle C(\bar{x}) = ce + \sum_{i=1}^{m} p_i \cdot C_i(\bar{x}_i), \varphi \rangle \in \mathcal{R}_P$, *the following cost equation formula holds*

$$\varphi \models f_C(\bar{x}) \geq ce + \sum_{i=1}^{m} p_i \cdot f_{C_i}(\bar{x}_i)$$

then $f_{C_{\ell_0}} \in F_{\mathcal{R}_P}$ *is an upper-bound on the expected cost of* P.

Proof. The proof follows from the definition of each case of transformation $\mathcal{T}$. It is immediate to see that each formula is an over-approximation of the corresponding case in the **ert** calculus of Figure 2.2. □

In what follows, for the sake of simplifying the presentation, we remove the equation of $C_\bullet(\bar{x})$ from $\mathcal{R}_P$ and replace the corresponding calls by 0. This clearly does not affect the expected cost.

EXAMPLE 3.2.8. Consider the cost relation of Example 3.2.4 and assume we are given functions $f_{C_{\ell_i}}(n)$ for each $C_{\ell_i} \in \texttt{crsym}(\mathcal{R}_P)$. The cost equation formulas that we should verify according to Theorem 3.2.7 are the following:

$$\begin{aligned}
[n \leq 0] &\models f_{C_{\ell_0}}(n) \geq 0 \\
[n > 0] &\models f_{C_{\ell_0}}(n) \geq f_{C_{\ell_1}}(n) \\
[\mathbf{n > 0}] &\models f_{C_{\ell_1}}(n) \geq 1 + f_{C_{\ell_2}}(n) \\
[\mathbf{n > 0}] &\models f_{C_{\ell_2}}(n) \geq \tfrac{3}{4} \cdot f_{C_{\ell_3}}(n) + \tfrac{1}{4} \cdot f_{C_{\ell_4}}(n) \\
[\mathbf{n > 0}, n' = n - 1] &\models f_{C_{\ell_3}}(n) \geq f_{C_{\ell_0}}(n') \\
[\mathbf{n > 0}] &\models f_{C_{\ell_4}}(n) \geq f_{C_{\ell_0}}(n)
\end{aligned}$$

Now let us take $f_{C_{\ell_0}}(n) = f_{C_{\ell_1}}(n) = \frac{4}{3} \cdot \|n\|$, $f_{C_{\ell_2}}(n) = \frac{4}{3} \cdot \|n - \frac{3}{4}\|$, $f_{C_{\ell_3}}(n) = \frac{4}{3} \cdot \|n - 1\|$, and $f_{C_{\ell_4}}(n) = \frac{4}{3} \cdot \|n\|$. Substituting them in the above formulas we get

$$\begin{aligned}
[n \leq 0] &\models \tfrac{4}{3} \cdot \|n\| \geq 0 \\
[n > 0] &\models \tfrac{4}{3} \cdot \|n\| \geq \tfrac{4}{3} \cdot \|n\| \\
[\mathbf{n > 0}] &\models \tfrac{4}{3} \cdot \|n\| \geq 1 + \tfrac{4}{3} \cdot \|n - \tfrac{3}{4}\|) \\
[\mathbf{n > 0}] &\models \tfrac{4}{3} \cdot \|n - \tfrac{3}{4}\| \geq \tfrac{3}{4} \cdot \tfrac{4}{3} \cdot \|n - 1\| + \tfrac{1}{4} \cdot \tfrac{4}{3} \cdot \|n\| \\
[\mathbf{n > 0}, n' = n - 1] &\models \tfrac{4}{3} \cdot \|n - 1\| \geq \tfrac{4}{3} \cdot \|n'\| \\
[\mathbf{n > 0}] &\models \tfrac{4}{3} \cdot \|n\| \geq \tfrac{4}{3} \cdot \|n\|
\end{aligned}$$

It is easy to verify that they hold, and thus $f_{C_{\ell_0}}(n) = \frac{4}{3} \cdot \|n\|$ is an upper-bound on the expected cost of the corresponding program. Note that without adding the invariant $\mathbf{n > 0}$ we would fail to find functions that satisfy the above formulas. □

Algorithm 1: Unfolding cost relations

```
   UNFOLDCRS(R)
   begin
1      A = {C_ℓ ∈ crsym(R) | ℓ is not a loop head or entry label }
2      foreach C_ℓ ∈ A do
3          Let R_1 ⊆ R be the set of all equations that define C_ℓ
4          R = R \ R_1
           while R has an equation calling C_ℓ do
5              Let R_2 ⊆ R be the set of all equations that call C_ℓ
6              R = R \ R_2
7              foreach E_1 ∈ R_2 do
8                  foreach E_2 ∈ R_1 do
9                      E = UNFOLD (E_1,E_2)
10                     R = R ∪ {E}
11     return R

   UNFOLD(E_1, E_2)
   begin
12     Let E_2 = ⟨D(ȳ) = ce_2 + Σ_{i=1}^{m_2} q_i · D_i(ȳ_i), φ_2⟩
13     Rewrite E_1 as ⟨C(x̄) = ce_1 + p · D(ȳ) + Σ_{i=1}^{m_1} p_i · C_i(x̄_i), φ_1⟩
14     E = ⟨C(x̄) = ce_1 + p · ce_2 + Σ_{i=1}^{m_2} p · q_i · D_i(ȳ_i) + Σ_{i=1}^{m_1} p_i · C_i(x̄_i), φ_1 ∪ φ_2⟩
15     return E
```

Theorem 3.2.7 give us a method to verify that a given set of functions are upper-bounds on the expected cost. In Section 3.4 we will see how to infer them automatically. In the meanwhile, in the next section, we describe how to simplify cost relations in order to reduce the number of cost relation symbols, and thus reduce the number of functions we need to verify (or infer).

3.3 Simplifying cost relations

Transforming probabilistic programs into cost relations as described in Figure 3.1 generates cost equations for all program points. This might affect the performance and precision when solving them into closed-form upper-bound functions because,

according to Theorem 2.3.2, we have to seek such functions for every cost relation symbol.

Our aim is to reduce the number of cost relation symbols in $\mathcal{R}_P$ using the techniques of [3]. The basic idea is based on inlining the definitions of some cost equations into their calling contexts. This operation is known as *unfolding*. Apart from reducing the number of cost relation symbols, it also has the effect of propagating (or more precisely grouping) constraints which in some cases eliminates the need for invariants.

Let us start by giving an intuition on how an equation is unfolded into another equation. Suppose E_1 is an equation that has a call to C_ℓ, and suppose that E_2 is an equation that defines C_ℓ. Unfolding E_2 into E_1 is done as follows:

1. Rename variables in E_1 such that the variables in *one of the calls* to C_ℓ are identical to those in the left-hand side of E_2, and all other variables are different from those of E_2.

2. Replace the call (of the step 1) to C_ℓ in E_1 by the right-hand side of E_2; and

3. Add the constraints of E_2 to those of E_1.

This intuition is depicted as procedure `UNFOLD`(E_1, E_2) in Algorithm 1, where D is the cost relation symbol that is unfolded.

Procedure `UNFOLDCRS` of Algorithm 1 uses `UNFOLD` to unfold all calls to cost relation symbols that are not loop heads or program entry. It first extracts the set A of cost relation symbols that do not correspond to loop heads or program entry, and then for each $C_\ell \in A$ it rewrites the rest of equations until no call to C_ℓ remains. Let us apply `UNFOLDCRS` to all the examples that we have discussed so far.

EXAMPLE 3.3.1. Consider the cost relation $\mathcal{R}_P$ of Example 3.2.4. The only cost relation symbol that corresponds to a loop head or program entry is C_{ℓ_0}. Applying

$\texttt{UNFOLDCRS}(\mathcal{R}_P)$ results in $\mathcal{R}'$:

$$\begin{array}{lll} C_{\ell_0}(n) = & 0 & [n \leq 0] \\ C_{\ell_0}(n) = & 1 + \frac{3}{4} \cdot C_{\ell_0}(n') + \frac{1}{4} \cdot C_{\ell_0}(n) & [n > 0,\ n' = n - 1] \end{array}$$

The equations defining C_{ℓ_1}, C_{ℓ_2}, C_{ℓ_3}, and C_{ℓ_4} were unfolded into C_{ℓ_0}. Generating the cost equation formulas as in Theorem 3.2.7 for the unfolded cost relation results in:

$$\begin{array}{rl} [n \leq 0] & \models f_{C_{\ell_0}}(n) \geq 0 \\ [n > 0, n' = n - 1] & \models f_{C_{\ell_0}}(n) \geq 1 + \frac{3}{4} \cdot f_{C_{\ell_0}}(n') + \frac{1}{4} \cdot f_{C_{\ell_0}}(n) \end{array}$$

The new $\mathcal{R}'$ has only 2 formulas, while the ones generated from $\mathcal{R}_P$ before unfolding in Example 3.2.8 included 6 formulas. It is easy to verify that the above formulas hold for $f_{C_{\ell_0}}(n) = \frac{4}{3} \cdot \|n\|$. Note that in this case, we could omit the invariant $\mathbf{n > 0}$ as it appears already in the definition of C_{ℓ_0}. □

EXAMPLE 3.3.2. Consider the cost relation $\mathcal{R}_P$ generated in Example 3.2.5. The only cost relation symbol that corresponds to a loop head or entry is C_{ℓ_0}. Applying $\texttt{UNFOLDCRS}(\mathcal{R}_P)$ results in:

$$\begin{array}{lll} C_{\ell_0}(n,m) = & 0 & [n \leq 0] \\ C_{\ell_0}(n,m) = & 0 & [m \leq 0] \\ C_{\ell_0}(n,m) = & 1 + \frac{3}{4} \cdot C_{\ell_0}(n',m) + \frac{1}{4} \cdot C_{\ell_0}(n,m) & [n > 0, m > 0, n' = n - 1] \\ C_{\ell_0}(n,m) = & 1 + \frac{3}{4} \cdot C_{\ell_0}(n,m') + \frac{1}{4} \cdot C_{\ell_0}(n,m) & [n > 0, m > 0, m' = m - 1] \end{array}$$

All equations have been unfolded into C_{ℓ_0}. Generating the cost equation formulas as in Theorem 3.2.7 for the unfolded cost relation results in:

$$\begin{array}{rl} [n \leq 0] & \models f_{C_{\ell_0}}(n,m) \geq 0 \\ [m \leq 0] & \models f_{C_{\ell_0}}(n,m) \geq 0 \\ [n > 0, m > 0, n' = n - 1] & \models f_{C_{\ell_0}}(n,m) \geq 1 + \frac{3}{4} \cdot f_{C_{\ell_0}}(n',m) + \frac{1}{4} \cdot f_{C_{\ell_0}}(n,m) \\ [n > 0, m > 0, m' = m - 1] & \models f_{C_{\ell_0}}(n,m) \geq 1 + \frac{3}{4} \cdot f_{C_{\ell_0}}(n,m') + \frac{1}{4} \cdot f_{C_{\ell_0}}(n,m) \end{array}$$

It is easy to verify that they hold for $f_{C_{\ell_0}}(n,m) = \frac{4}{3} \cdot \|n\| + \frac{4}{3} \cdot \|m\|$. □

EXAMPLE 3.3.3. Consider the cost relation $\mathcal{R}_P$ of Example 3.2.6. The only cost relation symbol that corresponds to a loop head or entry is C_{ℓ_0}. Applying

UNFOLDCRS($\mathcal{R}_P$) results in:

$$\begin{array}{lll} C_{\ell_0}(x,y,n,m) = & 0 & \varphi_1 \\ C_{\ell_0}(x,y,n,m) = & 1 + \frac{1}{2}\cdot C_{\ell_0}(x,y,n,m) + \frac{1}{2}\cdot C_{\ell_0}(x,y',n,m) & \varphi_2 \\ C_{\ell_0}(x,y,n,m) = & 1 + \frac{1}{4}\cdot C_{\ell_0}(x,y,n,m) + \sum_{i=1}^{3} \frac{3}{4}\cdot\frac{1}{3}\cdot C_{\ell_0}(x'_i,y,n,m) & \varphi_3 \end{array}$$

$\varphi_1 = [x+3 > n]$, $\varphi_2 = [x+3 \leq n, y < m, y' = y+1]$, and $\varphi_3 = [x+3 \leq n, y \geq m, x'_1 = x+1, x'_2 = x+2, x'_3 = x+3]$. Generating the cost equation formulas as in Theorem 3.2.7 for the unfolded cost relation results in:

$$\begin{array}{lll} \varphi_1 & \models & f_{C_{\ell_0}}(x,y,n,m) \geq 0 \\ \varphi_2 & \models & f_{C_{\ell_0}}(x,y,n,m) \geq 1 + \frac{1}{2}\cdot f_{C_{\ell_0}}(x,y,n,m) + \frac{1}{2}\cdot f_{C_{\ell_0}}(x,y',n,m) \\ \varphi_3 & \models & f_{C_{\ell_0}}(x,y,n,m) \geq 1 + \frac{1}{4}\cdot f_{C_{\ell_0}}(x,y,n,m) + \sum_{i=1}^{3} \frac{3}{4}\cdot\frac{1}{3}\cdot f_{C_{\ell_0}}(x'_i,y,n,m) \end{array}$$

It is easy to verify that they hold for $f_{C_{\ell_0}}(x,y,n,m) = \frac{2}{3}\cdot\|n-x\| + 2\cdot\|m-y\|$.

□

The following theorem states that unfolding is *sound* when considering the expected cost and, moreover, it is always *as precise as* the cost relation before unfolding.

THEOREM 3.3.4. *Let P be a program, $\mathcal{R}_P = \mathcal{T}(P, C_\bullet)$, and $\mathcal{R}'_P$ = UNFOLDCRS($\mathcal{R}_P$), then:* ***(soundness)*** *if F is a set of valid upper-bound functions for $\mathcal{R}'_P$, then $f_{C_{\ell_0}}$ is an upper-bound function on the expected cost of P; and* ***(precision)*** *if F is a set of valid upper-bound functions for $\mathcal{R}_P$, then it is also a set of valid upper-bound functions for $\mathcal{R}'_P$.*

Proof. Follows from the unfolding of cost relations of [3]. The difference from [3] is the handling of multiplication of calls by constants, but the proof in that paper extends naturally to this case. □

3.4 Solving cost relation formulas

In this section, we describe how to automatically find the set of functions $F_{\mathcal{R}_P}$ of Theorem 3.2.7, and thus an upper bound on the expected cost of the corresponding program. Let us start by addressing a simpler problem.

Let φ be a conjunction of the following (satisfiable) linear constraints

$$\begin{array}{lcl} a_{1,1} \cdot x_1 \quad + \cdots + \quad a_{1,n} \cdot x_n + c_1 & \geq & 0 \\ \vdots & & \\ a_{m,1} \cdot x_1 \quad + \cdots + \quad a_{m,n} \cdot x_n + c_m & \geq & 0 \end{array} \tag{3.2}$$

where $a_{i,j} \in \mathbb{Z}$, $c_i \in \mathbb{Z}$, $x_1, \ldots, x_n$ are variables, and let f be

$$f(\bar{x}) = d_0 + d_1 \cdot x_1 + \cdots + d_n \cdot x_n \tag{3.3}$$

where $d_0, \ldots, d_n$ are variable, not concrete numbers.

We call f a *template function* and $d_0, \ldots, d_n$ *template parameters*. Our goal is to instantiate f, i.e., find values for $d_0, \ldots, d_n$ such that the following formula holds:

$$\varphi \models d_1 \cdot x_1 + \cdots + d_n \cdot x_n + d_0 \geq 0 \tag{3.4}$$

In order to solve this problem, we make use of Farkas' Lemma [35] which tells us that formula (3.4) holds if and only if the following system of (linear) constraints is satisfiable:

$$\begin{array}{ll} \lambda_1 \geq 0, \ldots, \lambda_m \geq 0 & \\ d_0 \geq \sum_{i=0}^{m} \lambda_i \cdot c_i & \\ d_j = \sum_{i=0}^{m} \lambda_i \cdot a_{i,j} & \text{for } j \in [1..n] \end{array} \tag{3.5}$$

Here $\lambda_1, \ldots, \lambda_m$ are new variables. This means that we can use an SMT solver to obtain a solution for (3.5), and then take the values of $d_0, \ldots, d_n$ in this solution to define an instance of f that satisfies (3.4).

In (3.2) we can also allow using $=$ instead of $\geq$. In such case, if the ith row uses $=$, then we drop $\lambda_i \geq 0$ from (3.4). In addition, we can also allow using $>$ in (3.2) and handle it as follows. First note that all variables $x_1, \ldots, x_n$ in our context will be integer-valued variables since they correspond to program variables, and recall that all $a_{i,j}$ are integer. Handling $>$ consists of adding 1 to the right-hand side of the constraint and changing $>$ by $\geq$. For example, if we have a constraint $2 \cdot x + y > 2$, we change it to $2 \cdot x + y \geq 3$.

EXAMPLE 3.4.1. Suppose we are given $\varphi_1 = [x_1 \geq x_2, x_3 \leq x_1 + x_2]$ and we are asked to find $f(x_1, x_2, x_3) = d_0 + d_1 \cdot x_1 + d_2 \cdot x_2 + d_3 \cdot x_3$ such that $\varphi_1 \models f(x_1, x_2, x_3) \geq 0$.

Applying Farkas' Lemma we get the following system of linear constraints:

$$\begin{array}{rcl} \lambda_1 & \geq & 0 \\ \lambda_2 & \geq & 0 \\ d_0 & \geq & 0 \\ d_1 & = & \lambda_1 + \lambda_2 \\ d_2 & = & -\lambda_1 + \lambda_2 \\ d_3 & = & -\lambda_2 \end{array}$$

A possible trivial solution to the above constraints assigns 0 to all variables, which gives $f(x_1, x_2, x_3) = 0$, another solution is $d_0 = 1, d_1 = 1, d_2 = 1, d_3 = -1, \lambda_1 = 0$, and $\lambda_2 = 1$, which gives $f(x_1, x_2, x_3) = 1 + x_1 + x_2 - x_3$. □

Let us now use the method just described to solve a more general problem. Let $\varphi_1, \ldots, \varphi_k$ be conjunctions of linear constraints, and let $f_1, \ldots, f_n$ be *template functions* that might share template parameters between them, i.e., some use the same d_i as coefficients. Let ψ be a conjunction of linear constraints over the template parameters, e.g., requiring each template parameter to satisfy $d_i \geq 0$. Intuitively, ψ is used to restrict the values that template parameters can take. Our goal is to instantiate $f_1, \ldots, f_n$ such that the chosen values for the template parameters satisfy ψ, and all $\varphi_i \models f_i$ hold. This problem can be solved by computing (3.4) for each $\varphi_i \models f_i$, but using different $\lambda_1, \ldots, \lambda_m$ for each one, and then asking an SMT solver to find a solution that satisfies all instances of (3.4) and ψ.

EXAMPLE 3.4.2. Consider again φ_1 of Example 3.4.1, and let $\varphi_2 = [x_1 \geq 0, x_2 \leq 0, x_3 \geq x_1 + x_2]$, and $f(x_1, x_2, x_3) = d_0 + d_1 \cdot x_1 + d_2 \cdot x_2 + d_3 \cdot x_3$. We want to find values for d_0, d_1, d_2 and d_3 such that $\varphi_1 \models f(x_1, x_2, x_3) \geq 0$ and $\varphi_2 \models f(x_1, x_2, x_3) \geq 0$. The constraints (3.5) of $\varphi_1 \models f(x_1, x_2, x_3) \geq 0$ have been generated in Example 3.4.1, and

those of $\varphi_2 \models f(x_1, x_2, x_3) \geq 0$ are (using ξ_i instead of λ_i):

$$\begin{array}{rcl} \xi_1 & \geq & 0 \\ \xi_2 & \geq & 0 \\ \xi_3 & \geq & 0 \\ d_0 & \geq & 0 \\ d_1 & = & \xi_1 - \xi_3 \\ d_2 & = & -\xi_2 - \xi_3 \\ d_3 & = & \xi_3 \end{array}$$

Now pick a solution that satisfies this system of constraints and the one of Example 3.4.1. The solution that assigns 0 to all variables is still valid, but the other solution that we found in Example 3.4.1 is not valid since d_3 cannot be negative in the above constraints (it is equal to ξ_3 which is nonnegative). A possible solution is $d_0 = 1, d_1 = 1, d_2 = -1$ and $d_3 = 0$, which gives $f(x_1, x_2, x_3) = 1 + x_1 - x_2$. □

As a last generalization, note that in the *template functions* each d_i can be an arbitrary expression over some template parameters (variables different from $\bar{x}$). This does not require any change in the method described above. In what follows, the template parameters of a given template function f are defined as those variables different from $\bar{x}$.

Given a set of formulas

$$\Psi = \{\varphi_1 \models f_1(\bar{x}) \geq 0, \ldots, \varphi_k \models f_k(\bar{x}) \geq 0\}$$

where each φ_i is a conjunction of linear constraint over $\bar{x}$ and $f_1, \ldots, f_k$ are linear template functions, and a constraint ψ over the template parameters of $f_1, \ldots, f_k$, we denote by SolveFormulas(Ψ, ψ) a procedure that infers values for the template parameters using the method described above.

Now let us discuss how SolveFormulas can be used to solve the cost equation formulas of Theorem 3.2.7, i.e., synthesize upper-bound functions on the expected cost. The idea is to assign each $f_C(\bar{x})$ a template function, and then solve the corresponding formulas using SolveFormulas. However, this is not immediate since our

template functions make use of cost expressions of the form $\|l\|$ which are not linear since in Definition 2.2.1 we have $\|l\| = \max(0, l)$.

Assume a given cost relation $\mathcal{R}_P$ in the context of Theorem 3.2.7, and for each $C \in \texttt{crsym}(\mathcal{R}_P)$ let

$$f_C(\bar{x}) = d_0 + \sum_{i=0}^{k_C} d_i \cdot \|l_i\| \tag{3.6}$$

where each l_i is a (predefined) linear expression over variables $\bar{x}$ (as in Definition 2.2.1), and all d_i are template parameters (different functions can use different d_i and l_i). Now let us consider one cost equation formula from those generated in Theorem 3.2.7:

$$\varphi \models f_C(\bar{x}) \geq ce + \sum_{i=1}^{m} p_i \cdot f_{C_i}(\bar{x}_i) \tag{3.7}$$

If we substitute templates (3.6) in (3.7) we get a formula that can be written as follows:

$$\varphi \models \sum_i E_i \geq 0 \tag{3.8}$$

where each E_i is one of the following forms:

- a number;
- $p \cdot \|l\|$ where p is a number and l a linear expression over $\bar{x}$;
- $p \cdot d$ where p is a number and d is a template parameter from (3.6); or
- $p \cdot d \cdot \|l\|$ where p is a number, d is a template parameter from (3.6), and l a linear expression over $\bar{x}$.

The right-hand side of (3.8) is almost linear as the one of (3.4), the problem is the presence of expressions $\|l\|$ since they are not linear. Next, we see how to eliminate them.

Let us just concentrate on one E_i that has the form $X \cdot \|l\|$ where X is either p or $p \cdot b$:

$$\varphi \models \cdots + X \cdot \|l\| + \cdots \geq 0 \tag{3.9}$$

Since $\|l\| = \max(0, l)$, we can explicitly split (3.9) into two cases:

- One in which $l \geq 0$, and thus $\|l\|$ is equal to l.
- One in which $l < 0$, and thus $\|l\|$ is equal to 0.

These two cases can be represented as a conjunction of the two formulas:

$$\begin{aligned} \varphi \wedge l \geq 0 &\models \cdots + X \cdot \boxed{l} + \cdots \geq 0 \\ \varphi \wedge l < 0 &\models \cdots + X \cdot \boxed{0} + \cdots \geq 0 \end{aligned} \tag{3.10}$$

Note that the left-hand side of each formula includes only linear constraints over $\bar{x}$, since l is a linear expression over $\bar{x}$.

It is easy to see that (3.9) and (3.10) are equivalent and that in (3.10) we have eliminated the nonlinear expression E_i, that we started from. If we repeat this for each nonlinear expression $X \cdot \|l\|$, we can eliminate all nonlinear expressions and obtain formulas as in (3.4). Note that if the left-hand side of any of the formulas of (3.10) is *false*, then we drop the corresponding formula since it is valid by definition.

Let Ψ be the set of these formulas, and let ψ be a conjunction of linear constraints that requires all template parameters to be nonnegative, then we can instantiate the template functions (3.6) by calling SolveFormulas(Ψ, ψ).

EXAMPLE 3.4.3. Consider the cost equation formulas of Example 3.3.1. Substituting the template $f_{C_{\ell_0}}(n) = d_0 + d_1 \cdot \|n\|$ we get

$$\begin{array}{lll} [n \leq 0] & \models & d_0 + d_1 \cdot \|n\| \geq 0 \\ [n > 0] & \models & d_0 + d_1 \cdot \|n\| \geq 1 + \frac{3}{4} \cdot (d_0 + d_1 \cdot \|n'\|) + \frac{1}{4} \cdot (d_0 + d_1 \cdot \|n\|) \end{array}$$

and simplifying the right-hand sides we get:

$$\begin{array}{lll} [n \leq 0] & \models & d_0 + d_1 \cdot \|n\| \geq 0 \\ [n > 0] & \models & \frac{3}{4} \cdot d_1 \cdot \|n\| - \frac{3}{4} \cdot d_1 \cdot \|n'\| - 1 \geq 0 \end{array}$$

Splitting the nonlinear terms $\|n\|$ and $\|n'\|$ we get:

$$\begin{array}{rl} [n \leq 0, \underline{n \geq 0}] & \models d_0 + d_1 \cdot \boxed{n} \geq 0 \\ [n \leq 0, \underline{n < 0}] & \models d_0 + d_1 \cdot \boxed{0} \geq 0 \end{array}$$

$$\begin{array}{rl} [n > 0, n' = n - 1, \underline{n \geq 0, n' \geq 0}] & \models \frac{3}{4} \cdot d_1 \cdot \boxed{n} - \frac{3}{4} \cdot d_1 \cdot \boxed{n'} - 1 \geq 0 \\ [n > 0, n' = n - 1, \underline{n \geq 0, n' < 0}] & \models \frac{3}{4} \cdot d_1 \cdot \boxed{n} - \frac{3}{4} \cdot d_1 \cdot \boxed{0} - 1 \geq 0 \\ [n > 0, n' = n - 1, \underline{n < 0, n' \geq 0}] & \models \frac{3}{4} \cdot d_1 \cdot \boxed{0} - \frac{3}{4} \cdot d_1 \cdot \boxed{n'} - 1 \geq 0 \\ [n > 0, n' = n - 1, \underline{n < 0, n' < 0}] & \models \frac{3}{4} \cdot d_1 \cdot \boxed{0} - \frac{3}{4} \cdot d_1 \cdot \boxed{0} - 1 \geq 0 \end{array}$$

The underlined constraints are those added due to splitting. The left-hand sides of the last three formulas are unsatisfiable, and thus the corresponding formulas can be ignored. Let us denote the first three formulas by Ψ. Calling SolveFormulas(Ψ, $[d_0 \geq 0, d_1 \geq 0]$) first generates the constraints

$\lambda_{1,1} \geq 0$	$\lambda_{2,1} \geq 0$	$\lambda_{3,1} \geq 0$
$\lambda_{1,2} \geq 0$	$\lambda_{2,2} \geq 0$	$\lambda_{3,3} \geq 0$
		$\lambda_{3,4} \geq 0$
$-\lambda_{1,1} + \lambda_{1,2} = d_1$	$-\lambda_{2,1} - \lambda_{2,2} = 0$	$\lambda_{3,1} - \lambda_{3,2} + \lambda_{3,3} = \frac{3}{4} \cdot d_1$
$d_0 \geq 0$	$d_0 \geq -\lambda_{2,2}$	$\lambda_{3,2} + \lambda_{3,4} = -\frac{3}{4} \cdot d_1$
		$-1 \geq -\lambda_{3,1} + \lambda_{3,2}$

Then, while solving them using an SMT solver, we obtain a solution in which $d_0 = 0$ and $d_1 = \frac{4}{3}$, and thus $f_{C_{\ell_0}}(n) = \frac{4}{3} \cdot \|n\|$ is an upper-bound function on the expected cost of the corresponding program. □

EXAMPLE 3.4.4. Consider the cost equation formulas of Example 3.3.2. Substituting the template $f_{C_{\ell_0}}(n, m) = d_0 + d_1 \cdot \|n\| + d_2 \cdot \|m\|$ and simplifying the right-hand sides we get:

$$\begin{array}{rcl} [n \leq 0] & \models & d_0 + d_1 \cdot \|n\| + d_2 \cdot \|m\| \geq 0 \\ [m \leq 0] & \models & d_0 + d_1 \cdot \|n\| + d_2 \cdot \|m\| \geq 0 \\ [n > 0, m > 0, n' = n - 1] & \models & \frac{3}{4} \cdot d_1 \cdot \|n\| - \frac{3}{4} \cdot d_1 \cdot \|n'\| - 1 \geq 0 \\ [n > 0, m > 0, m' = m - 1] & \models & \frac{3}{4} \cdot d_2 \cdot \|m\| - \frac{3}{4} \cdot d_2 \cdot \|m'\| - 1 \geq 0 \end{array}$$

Splitting the nonlinear terms results on the following (formulas with unsatisfiable

left-hand side have been removed)

$$\begin{array}{rl}
[n \le 0, \underline{n \ge 0, m \ge 0}] & \models d_0 + d_1 \cdot \boxed{n} + d_2 \cdot \boxed{m} \ge 0 \\
[n \le 0, \underline{n < 0, m \ge 0}] & \models d_0 + d_1 \cdot \boxed{0} + d_2 \cdot \boxed{m} \ge 0 \\
[n \le 0, \underline{n \ge 0, m < 0}] & \models d_0 + d_1 \cdot \boxed{n} + d_2 \cdot \boxed{0} \ge 0 \\
[n \le 0, \underline{n < 0, m < 0}] & \models d_0 + d_1 \cdot \boxed{0} + d_2 \cdot \boxed{0} \ge 0
\end{array}$$

$$\begin{array}{rl}
[m \le 0, \underline{n \ge 0, m \ge 0}] & \models d_0 + d_1 \cdot \boxed{n} + d_2 \cdot \boxed{m} \ge 0 \\
[m \le 0, \underline{n < 0, m \ge 0}] & \models d_0 + d_1 \cdot \boxed{0} + d_2 \cdot \boxed{m} \ge 0 \\
[m \le 0, \underline{n \ge 0, m < 0}] & \models d_0 + d_1 \cdot \boxed{n} + d_2 \cdot \boxed{0} \ge 0 \\
[m \le 0, \underline{n < 0, m < 0}] & \models d_0 + d_1 \cdot \boxed{0} + d_2 \cdot \boxed{0} \ge 0
\end{array}$$

$$[n > 0, m > 0, n' = n - 1, \underline{n \ge 0, n' \ge 0}] \models \tfrac{3}{4} \cdot d_1 \cdot \boxed{n} - \tfrac{3}{4} \cdot d_1 \cdot \boxed{n'} - 1 \ge 0$$

$$[n > 0, m > 0, m' = m - 1, \underline{m \ge 0, m' \ge 0}] \models \tfrac{3}{4} \cdot d_2 \cdot \boxed{m} - \tfrac{3}{4} \cdot d_2 \cdot \boxed{m'} - 1 \ge 0$$

Let us denote these formulas by Ψ, calling SolveFormulas($\Psi, [d_0 \ge 0, d_1 \ge 0, d_2 \ge 0]$) instantiates the template upper-bound function to $f_{C_{\ell_0}} = \frac{4}{3} \cdot \|n\| + \frac{4}{3} \cdot \|m\|$. □

EXAMPLE 3.4.5. Consider the cost equation formulas of Example 3.3.3, substituting the template $f_{C_{\ell_0}}(n, m, x, y) = d_0 + d_1 \cdot \|n - x\| + d_2 \cdot \|m - y\|$, and simplifying the right-hand sides we get the equations

$$\begin{array}{rcl}
\varphi_1 & \models & d_0 + d_1 \cdot \|n - x\| + d_2 \cdot \|m - y\| \ge 0 \\
\varphi_2 & \models & \frac{1}{2} \cdot d_2 \cdot \|m - y\| - \frac{1}{2} \cdot d_2 \cdot \|m - y'\| - 1 \ge 0 \\
\varphi_3 & \models & \frac{3}{4} \cdot d_1 \cdot \|n - x\| - \frac{1}{4} \cdot d_1 \cdot \|n - x'_1\| - \frac{1}{4} \cdot d_1 \cdot \|n - x'_2\| \\
 & & -\frac{1}{4} \cdot d_1 \cdot \|n - x'_3\| - 1 \ge 0
\end{array}$$

where $\varphi_1 = [x + 3 > n]$, $\varphi_2 = [x + 3 \le n, y < m, y' = y + 1]$, and $\varphi_3 = [x + 3 \le n, y \ge m, x'_1 = x + 1, x'_2 = x + 2, x'_3 = x + 3]$. Splitting of nonlinear terms of the second formula results in:

$$[x + 3 \le n, y < m, y' = y + 1, \underline{m - y \ge 0, m - y' \ge 0}]$$

$$\models \tfrac{1}{2} \cdot d_2 \cdot \boxed{(m - y)} - \tfrac{1}{2} \cdot d_2 \cdot \boxed{(m - y')} - 1 \ge 0$$

Let us denote the formula together with the others that we do not show by Ψ. Calling SolveFormulas($\Psi, [d_0 \ge 0, d_1 \ge 0, d_2 \ge 0]$) instantiates the upper-bound function template to $f_{C_{\ell_0}} = \frac{2}{3} \cdot \|n - x\| + 2 \cdot \|m - y\|$. □

3.5 Automatic inference of template functions

In order to achieve full automation, what is left is to develop techniques for inferring template functions automatically. For this, we use heuristics that extract such functions from the conditions used in the program.

Recall that we are dealing with cost relations after applying `UNFOLDCRS` to unfold the cost relation, and thus every $C_\ell \in \texttt{crsym}(\mathcal{R}_P)$ corresponds to a loop head or program entry labeled with ℓ. For each such ℓ, we first compute a set T_ℓ of homogeneous linear expressions (i.e., the free constant is 0) as follows. If b is a condition used in the program (either in **if** or in **while**) and it is reachable from the program point corresponding to ℓ then:

- If b is of the form $l_1 \geq l_2$ or $l_1 > l_2$, where l_i is a linear expression, we add $l_1 - l_2$ to T_ℓ after removing the free constant.
- If b is of the form $l_1 \leq l_2$ or $l_1 < l_2$, where l_i is a linear expression, we add $l_2 - l_1$ to T_ℓ after removing the free constant.
- If b is of the form $l_1 == l_2$ or $l_1 =! l_2$, where l_i is a linear expression, we add $l_1 - l_2$ and $l_2 - l_1$ to T_ℓ after removing the free constant.

Then, assuming $T_\ell = \{l_1, \ldots, l_k\}$, we define the template function f_{C_ℓ} of $C_\ell \in \texttt{crsym}(\mathcal{R}_P)$ as

$$f_{C_\ell}(\bar{x}) = d_0 + \sum_{i=0}^{k} d_i \cdot \|l_i\| \tag{3.11}$$

EXAMPLE 3.5.1.

- For the cost relation of Example 3.2.4, the condition $n > 0$ is reachable from ℓ_0, therefore $T_{\ell_0} = \{n\}$ and $f_{C_{\ell_0}} = d_0 + d_1 \cdot \|n\|$.
- For the cost relation of Example 3.2.5, the conditions $n > 0$ and $m > 0$ are both are reachable from ℓ_0, therefore $T_{\ell_0} = \{n, m\}$ and $f_{C_{\ell_0}} = d_0 + d_1 \cdot \|n\| + d_2 \cdot \|m\|$.

- For the cost relation of Example 3.2.6, the conditions $x + 3 \leq n$ and $y < m$ are both reachable from ℓ_0, therefore $T_\ell = \{n - x, m - y\}$ and $f_{C_{\ell_0}} = d_0 + d_1 \cdot \|n - x\| + d_2 \cdot \|m - y\|$. Note that we have used $n - x$ because it is what we obtain by removing the free constant from $n - x - 3$.

□

Let us end this chapter with an example that has more than one loop.

EXAMPLE 3.5.2. Consider the following program:

```
while (n>0)^ℓ0 {
  tick(1)^ℓ3;
  n :=^ℓ5 n−1; ⊕^ℓ4_(3/4) skip^ℓ6;
}
tick(8)^ℓ1;
while(m>0)^ℓ2 {
        tick(2)^ℓ7;
        m :=^ℓ8 m−1;
}
```

Generating the cost relation for this program and then applying `UNFOLDCRS` results in the following cost relation:

$$\begin{array}{lll} C_{\ell_0}(n,m) = & 1 + \frac{3}{4} \cdot C_{\ell_0}(n',m) + \frac{1}{4} \cdot C_{\ell_0}(n,m) & [n > 0,\ n' = n - 1] \\ C_{\ell_0}(n,m) = & 8 + C_{\ell_2}(n,m) & [n \leq 0] \\ C_{\ell_2}(n,m) = & 2 + C_{\ell_2}(n,m') & [m > 0,\ m' = m - 1] \\ C_{\ell_2}(n,m) = & 0 & [m \leq 0] \end{array}$$

Here, C_{ℓ_0} models the expected cost of the first loop (and its continuation), and C_{ℓ_2} models the expected cost of the second loop. Note that in the case of C_{ℓ_0} we accumulate 8 (which corresponds to **tick(8)**), and call C_{ℓ_2} in order to account for the cost of the second loop.

Next, we generate the templates for C_{ℓ_0} and C_{ℓ_2}. Since conditions $n > 0$ and $m > 0$ are reachable from ℓ_0 we get $T_{\ell_0} = \{n, m\}$ and $f_{C_{\ell_0}} = d_0 + d_1 \cdot \|n\| + d_2 \cdot \|m\|$, and since only $m > 0$ is reachable from ℓ_1 we get $T_{\ell_2} = \{m\}$ and $f_{C_{\ell_2}} = d'_0 + d'_1 \cdot \|m\|$.

Using these templates, the cost equation formulas generated by Theorem 3.2.7, after some simplification, are:

$$\begin{array}{rcl}
[n > 0,\ n' = n - 1] & \models & \frac{3}{4} \cdot d_1 \cdot \|n\| - \frac{3}{4} \cdot d_1 \cdot \|n'\| - 1 \geq 0 \\
[n \leq 0] & \models & d_1 \cdot \|n\| + (d_2 - d_1') \cdot \|m\| + d_0 - d_0' - 8 \geq 0 \\
[m > 0,\ m' = m - 1] & \models & d_1' \cdot \|m\| - d_1' \cdot \|m'\| - 2 \geq 0 \\
[m \leq 0] & \models & d_1' \cdot \|m\| + d_0' \geq 0
\end{array}$$

Then, splitting the nonlinear expressions $\|n\|$,$\|n'\|$,$\|m\|$,$\|m'\|$, we get (after removing those whose left-hand side is *false*):

$$[n > 0,\ n' = n - 1,\ \underline{n \geq 0,\ n' \geq 0}] \models \frac{3}{4} \cdot d_1 \cdot \boxed{n} - \frac{3}{4} \cdot d_1 \cdot \boxed{n'} - 1 \geq 0$$

$$\begin{array}{rcl}
[n \leq 0,\ \underline{n \geq 0,\ m \geq 0}] & \models & d_1 \cdot \boxed{n} + (d_2 - d_1') \cdot \boxed{m} + d_0 - d_0' - 8 \geq 0 \\
[n \leq 0,\ \underline{n < 0,\ m \geq 0}] & \models & d_1 \cdot \boxed{0} + (d_2 - d_1') \cdot \boxed{m} + d_0 - d_0' - 8 \geq 0 \\
[n \leq 0,\ \underline{n \geq 0,\ m < 0}] & \models & d_1 \cdot \boxed{n} + (d_2 - d_1') \cdot \boxed{0} + d_0 - d_0' - 8 \geq 0 \\
[n \leq 0,\ \underline{n < 0,\ m < 0}] & \models & d_1 \cdot \boxed{0} + (d_2 - d_1') \cdot \boxed{0} + d_0 - d_0' - 8 \geq 0
\end{array}$$

$$[m > 0,\ m' = m - 1,\ \underline{m \geq 0,\ m' \geq 0}] \models d_1' \cdot \boxed{m} - d_1' \cdot \boxed{m'} - 2 \geq 0$$

$$\begin{array}{rcl}
[m \leq 0,\ \underline{m \geq 0}] & \models & d_1' \cdot \boxed{m} + d_0' \geq 0 \\
[m \leq 0,\ \underline{m < 0}] & \models & d_1' \cdot \boxed{0} + d_0' \geq 0
\end{array}$$

Now let Ψ be the set of these formulas, and let ψ be a conjunction of linear constraints that requires all template parameters to be nonnegative. Then, when calling SolveFormulas(Ψ, ψ) we instantiate the template functions to:

$$\begin{array}{rl}
f_{C_{\ell_0}} & = \frac{4}{3} \cdot \|n\| + 2 \cdot \|m\| + 8 \\
f_{C_{\ell_2}} & = 2 \cdot \|m\|
\end{array}$$

□

CHAPTER 4

Implementation

The techniques presented in this work have been implemented in `Python`, and the final result is an analyzer that receives a probabilistic program as input and generates closed-form upper-bound functions on the expected cost. The source code of the analyzer can be downloaded from this link:

`https://drive.google.com/open?id=1NtTbQbZbStULP-J2fXgUV_wgv5z13kC_`

In what follows we describe the syntax of programs accepted by the analyzer, discuss its workflow, and give some usage examples.

4.1 Syntax of probabilistic programs

Programs accepted by the analyzer are written in a syntax that is very similar to the one described Section 2.2.1. The only difference is the use of curly brackets for blocks of codes (as in languages like `Java` or `C`), which simplify parsing, and the use a text-based representation for the following instructions:

- id := e + **unif**(a..b): probabilistic assignment (with uniform distribution) is written as "id := e ++ [a,b]"

- $c_1 \oplus_{a/b} c_n$: probabilistic branching is written as "c_1 .+ a/b c_2".
- $c_1 \diamond c_2$: nondeterministic choice is written as "c_1 <> c_2".

Using this syntax, the examples that we have used in the previous chapters are written as follows:

Example 3.2.4:

```
while (n >= 0) {
  tick(1);
  n := n-1; .+ 3/4 skip;
}
```

Example 3.2.5:

```
while (n > 0 and m > 0) {
  tick(1);
    (n := n-1; <> m := m-1;)
  .+ 3/4
    skip;
}
```

Example 3.2.6:

```
while (x+3 <= n) {
  if(y<m) then
    skip; .+ 1/2 y:= y+1;
  else
    skip; .+ 1/4 x:= x ++ [1,3];
  tick(1);
}
```

Example 3.5.2:

```
while (n > 0) {
  tick(1);
  n := n-1; .+ 3/4 skip;
}
tick(8);
while (m > 0) {
  tick(2);
  m := m-1;
}
```

4.2 Workflow of the analyzer

The analyzer includes two main components: (1) *translation*, which is responsible on translating a given program into a cost relation (including unfolding) and inference of template functions; and (2) *solving*, which solves the cost relation into closed-form upper-bound functions on the expected cost.

The workflow of the *translation* component is as follows:

- It parses the input program, which is done by using the `LARK` package [1] and generates an abstract syntax tree.
- It generates a cost relation from the abstract syntax tree as described in Section 3.2.
- It unfolds the cost relation as described in Section 3.3.
- It generates template functions as described in Section 3.5.

The *solving* component receives the cost relation and templates functions that are generated by the *translation* component, and proceeds as follow:

- It generates the set of cost equation formulas as described in Theorem 3.2.7.
- It solves the cost equation formulas using a procedure that implements the functionality of SolveFormulas as described in Section 3.4. For solving the generated system of linear constraints we use the SMT solver `Z3`[2] [16].

Note that the *solving* component can be used as a standalone tool as well, i.e., it can solve cost relations that are not necessarily generated by the first component.

4.3 Examples of execution

The following are some usage examples of the analyzer

[1] `https://github.com/lark-parser/lark`, licensed under the MIT License
[2] Licensed under the MIT License

Example 3.2.4:

```
$ python expcost.py input1

f_A(n)=+4/3*max(0,n)
```

Example 3.2.5:

```
$ python expcost.py input2

f_A(n,m)=+4/3*max(0,m)+4/3*max(0,n)
```

Example 3.2.6:

```
$ python expcost.py input3

f_A(x,n,y,m)=+2/3*max(0,-x+n)
      +2*max(0,-y+m)
```

Example 3.5.2:

```
$ python expcost.py input4

f_A(n,m)=+2*max(0,m)+8*max(0,1)
      +4/3*max(0,n)
```

CHAPTER 5

Conclusions

In this work, we have addressed the problem of automatically inferring closed-form upper-bound functions on the expected cost of probabilistic programs. The expected cost is different from the classical notion of worst-case cost in that it takes the probabilities of each execution into account. Thus, it is more adequate for programs that involve probabilistic choices.

Our approach followed a methodology used before for developing tools for worst-case cost analysis, which starts by exploring the limitations of using classical recurrence equations, extending them to what is called cost relations to handle programs with complex control-flow and nondeterminism, and develop techniques for solving them into closed-form upper-bound functions. In particular, we have extended the definition of cost relations to allow incorporating probabilities, developed a transformation that translates probabilistic programs into such cost relations that model their expected cost, and developed techniques for solving these cost relations into closed-form upper-bound functions. Our solving techniques are based on the use of linear programming [35]. We note that same techniques work also for inferring lower-bounds on the expected cost, simply by replacing $\geq$ to $\leq$ in Theorem 3.2.7.

When compared to the state-of-the-art tools [33, 42], our techniques cannot infer polynomial bounds as they do, which we leave for future work. However, our technique is simpler and can handle classical programs that appear in related literature.

5.1 Related work

Over the past decade several cost analysis frameworks, for different programming languages, have been developed [10, 24, 29, 38, 37, 4, 26, 6, 21, 20, 17, 32, 18, 36, 43]. They can infer precise upper-bounds on the *worst-case* cost and lower-bounds on the *best-case* cost. Early work on cost analysis [43, 25, 41, 34, 17, 23, 23, 9, 40] started by automating the classical technique used in (manual) complexity analysis which models the cost of a program using recurrence equations and then solves them into closed-form functions using off-the-shelf computer algebra systems. Cost relations were introduced in [3] as a generalization of recurrence equations to allow for nondeterminism. Several practical techniques for solving cost relations into closed-forms functions have been developed [3, 6, 21, 20]. There are other cost analysis techniques that are based on amortized analysis [37, 26, 20], invariants generation [24], and alternation of inferring size and complexity bounds [10].

Automatic *expected cost* analysis for probabilistic programs is relatively a new research field and has been recently considered in several works [28, 33, 7, 13, 42]. A breakthrough that triggered practical research in this field is part of [28], where the expected cost was formalized using a weakest precondition calculus that is able to handle nondeterministic programs as well, which was a major difficulty until then. As we have mentioned before, nondeterminism is very common in practice since the analyzed programs are typically the result of abstractions that introduce nondeterminism in order to abstract away from the particularities of a specific programming language. Thus, restricting ourselves to deterministic programs would reduce the applicability of the corresponding tools. Recently, the weakest preconditions calculus

of [28] has been used in [33] to develop a cost analyzer that is able to infer polynomial bounds on the expected cost of challenging problems. Other works in this field concentrated on different programming models such as term rewrite systems [7], on the use of recurrence relation [13], and on handling negative and unbounded cost [42].

There is also a considerable amount of work on what is called *almost sure* termination [1, 22, 12, 14, 15, 19, 11], which guarantees termination with probability 1. It is important to note that when counting loop iterations (or execution steps), boundedness of the expected cost implies *almost sure* termination, but *almost sure* termination does not imply that the expected cost is bounded. For example, consider the following program [28]:

```
x=1;
while ( n > 0 ) {
  n = 0; ⊕_{1/2} n=1;
  x = 2*x;
}
while (x>0) {
  x = x−1;
  tick(1);
}
```

It terminates with probability 1 since the expected number of iterations of the first loop is $\sum_{i=0}^{\infty}(\frac{1}{2})^i = 2$. However, the expected cost is not bounded since it is equal to the sum $\sum_{i=0}^{\infty}(\frac{1}{2})^i \cdot 2^i$ which is ∞. Here, $(\frac{1}{2})^i$ is the probability that the first loop makes i iterations, and 2^i is the value of x after the first loop which is also the cost of the corresponding trace since the second loop decreases x until it reaches 0.

5.2 Future work

We have identified several research directions that we would like to explore in future work. The first direction, which we referred to as the long term goal in Chapter 1, deals with the integration of our preliminary implementation in the static

analyzer SACO [2], which currently infers upper-bounds on the worst-case cost of ABS programs [27] — an abstract behavior modeling language based on concurrent objects. This includes the modification of the ABS language to include probabilistic instructions similar to those of our language, which will allow modeling the behavior of problems that use probabilities. The second direction is related to extending our approach to infer polynomial bounds, which would require the development of advanced techniques for solving cost relations (extending those of Section 3.4 as well). The third direction is modularity of the analysis, which is a challenging problem since the expected cost is not compositional.

Bibliography

[1] Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. *PACMPL*, 2(POPL):34:1–34:32, 2018.

[2] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, German Puebla, and Guillermo Román-Díez. SACO: static analyzer for concurrent objects. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 562–567. Springer, 2014.

[3] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-form upper bounds in static cost analysis. *J. Autom. Reasoning*, 46(2):161–203, 2011.

[4] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.

[5] Elvira Albert, Samir Genaim, and Abu Naser Masud. On the inference of resource usage upper and lower bounds. *ACM Trans. Comput. Log.*, 14(3):22:1–22:35, 2013.

[6] Diego Esteban Alonso-Blas, Puri Arenas, and Samir Genaim. Precise cost analysis via local reasoning. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2013.

[7] Martin Avanzini, Ugo Dal Lago, and Akihisa Yamada. On probabilistic term rewriting. In John P. Gallagher and Martin Sulzmann, editors, *Functional and Logic Programming - 14th International Symposium, FLOPS 2018, Nagoya, Japan, May 9-11, 2018, Proceedings*, volume 10818 of *Lecture Notes in Computer Science*, pages 132–148. Springer, 2018.

[8] Roberto Bagnara, Andrea Pescetti, Alessandro Zaccagnini, and Enea Zaffanella. PURRS: towards computer algebra support for fully automatic worst-case complexity analysis. *CoRR*, abs/cs/0512056, 2005.

[9] Ralph Benzinger. Automated complexity analysis of nuprl extracted programs. *Journal of Functional Programming*, 11(1):3–31, 2001.

[10] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.*, 38(4):13:1–13:50, 2016.

[11] Aleksandar Chakarov and Sriram Sankaranarayanan. Probabilistic program analysis with martingales. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Pe-*

tersburg, Russia, July 13-19, 2013. Proceedings, volume 8044 of *Lecture Notes in Computer Science*, pages 511–526. Springer, 2013.

[12] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Termination analysis of probabilistic programs through positivstellensatz's. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2016.

[13] Krishnendu Chatterjee, Hongfei Fu, and Aniket Murhekar. Automated recurrence analysis for almost-linear expected-runtime bounds. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 118–139. Springer, 2017.

[14] Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. *ACM Trans. Program. Lang. Syst.*, 40(2):7:1–7:45, 2018.

[15] Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. Stochastic invariants for probabilistic termination. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 145–160. ACM, 2017.

[16] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the*

Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[17] Saumya K. Debray and Nai-Wei Lin. Cost analysis of logic programs. *ACM Trans. Program. Lang. Syst.*, 15(5):826–875, 1993.

[18] Saumya K. Debray, Pedro López-García, Manuel V. Hermenegildo, and Nai-Wei Lin. Lower bound cost estimation for logic programs. In Jan Maluszynski, editor, *Logic Programming, Proceedings of the 1997 International Symposium, Port Jefferson, Long Island, NY, USA, October 13-16, 1997*, pages 291–305. MIT Press, 1997.

[19] Luis María Ferrer Fioriti and Holger Hermanns. Probabilistic termination: Soundness, completeness, and compositionality. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 489–501. ACM, 2015.

[20] Antonio Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 254–273, 2016.

[21] Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In Jacques Garrigue, editor, *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-*

19, 2014, Proceedings, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, 2014.

[22] Hongfei Fu and Krishnendu Chatterjee. Termination of nondeterministic probabilistic programs. In Constantin Enea and Ruzica Piskac, editors, *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, volume 11388 of *Lecture Notes in Computer Science*, pages 468–490. Springer, 2019.

[23] Bernd Grobauer. Cost recurrences for dml programs. In Benjamin C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, pages 253–264. ACM, 2001.

[24] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 127–139. ACM, 2009.

[25] Timothy J. Hickey and Jacques Cohen. Automating program analysis. *Journal of the ACM*, 35(1):185–220, 1988.

[26] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14:1–14:62, 2012.

[27] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A Core Language for Abstract Behavioral Specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*,

volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2012.

[28] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected runtimes of randomized algorithms. *J. ACM*, 65(5):30:1–30:68, 2018.

[29] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas W. Reps. Compositional recurrence analysis revisited. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 248–262. ACM, 2017.

[30] Mathematica: the Way the World Calculates. `http://www.wolfram.com/products/mathematica/index.html`.

[31] Alicia Merayo and Samir Genaim. Inference of linear upper-bounds on the expected cost by solving cost relations. In Salvador Lucas, editor, *Proceedings of the 16th International Workshop on Termination, WST'18 Oxford*, pages 60–64, 2018.

[32] Jorge A. Navas, Edison Mera, Pedro López-García, and Manuel V. Hermenegildo. User-definable resource bounds analysis for logic programs. In Verónica Dahl and Ilkka Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings*, volume 4670 of *Lecture Notes in Computer Science*, pages 348–363. Springer, 2007.

[33] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: resource analysis for probabilistic programs. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Pro-*

gramming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018, pages 496–512. ACM, 2018.

[34] Mads Rosendahl. Automatic complexity analysis. In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 144–156, 1989.

[35] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1986.

[36] Alejandro Serrano, Pedro López-García, and Manuel V. Hermenegildo. Resource usage analysis of logic programs via abstract interpretation using sized types. *TPLP*, 14(4-5):739–754, 2014.

[37] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 745–761. Springer, 2014.

[38] Moritz Sinn, Florian Zuleger, and Helmut Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *J. Autom. Reasoning*, 59(1):3–45, 2017.

[39] Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.*, 32(3):8:1–8:70, 2010.

[40] Pedro B. Vasconcelos and Kevin Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In Philip W. Trinder,

Greg Michaelson, and Ricardo Peña, editors, *Implementation of Functional Languages, 15th International Workshop, IFL 2003, Edinburgh, UK, September 8-11, 2003, Revised Papers*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2004.

[41] Philip Wadler. Strictness analysis aids time analysis. In Jeanne Ferrante and P. Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 119–132. ACM, 1988.

[42] Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. Cost analysis of nondeterministic probabilistic programs. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, pages 204–220. ACM, 2019.

[43] Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

[44] Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993.